ВЫСШЕЕ

ОБРАЗОВАНИЕ

В. В. Трофимов, Т. А. Павловская

Под редакцией В. В. Трофимова

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

Учебник
4-е издание

Курс с on-line оцениванием

УМО ВО
РЕКОМЕНДУЕТ

ЮРАЙТ
ИЗДАТЕЛЬСТВО

В. В. Трофимов, Т. А. Павловская

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

УЧЕБНИК ДЛЯ ВУЗОВ

Под редакцией **В. В. Трофимова**

4-е издание

Рекомендовано Учебно-методическим отделом высшего образования в качестве учебника для студентов высших учебных заведений, обучающихся по экономическим направлениям

Курс с практическими заданиями и дополнительными материалами доступен на образовательной платформе «Юрайт», а также в мобильном приложении «Юрайт.Библиотека»

Москва ▪ Юрайт ▪ 2024

УДК 681.3(075.8)
ББК 32.81я73
Т76

Авторы:

Трофимов Валерий Владимирович — доктор технических наук, профессор, заслуженный деятель науки Российской Федерации, заведующий кафедрой информатики факультета информатики и прикладной математики Института экономики Санкт-Петербургского государственного экономического университета, профессор кафедры информатики и прикладной математики — 1 факультета компьютерных технологий и управления Санкт-Петербургского национального исследовательского университета информационных технологий, механики и оптики.

Павловская Татьяна Александровна — профессор, кандидат технических наук, работала в должности профессора на кафедре информатики Санкт-Петербургского государственного экономического университета.

Трофимов, В. В.

Т76 Алгоритмизация и программирование : учебник для вузов / В. В. Трофимов, Т. А. Павловская. — 4-е изд. — Москва : Издательство Юрайт, 2024. — 108 с. — (Высшее образование). — Текст : непосредственный.

ISBN 978-5-534-20430-8

В курсе, представляющем собой один из модулей дисциплины «Информатика», рассмотрены модели решения функциональных и вычислительных задач, алгоритмизация и программирование, языки программирования высокого уровня, технологии программирования.

Курс соответствует актуальным требованиям федерального государственного образовательного стандарта высшего образования.

Для студентов высших учебных заведений, обучающихся по экономическим направлениям, аспирантов и преподавателей, специалистов организаций любого уровня и сферы хозяйствования.

УДК 681.3(075.8)
ББК 32.81я73

ISBN 978-5-534-20430-8

Оглавление

ТЕМА 1
ОСНОВЫ АЛГОРИТМИЗАЦИИ

1.1. Понятие алгоритма и его свойства

Алгоритмом называют точное предписание, которое задается вычислительному процессу и представляет собой конечную последовательность обычных элементарных действий, четко определяющую процесс преобразования исходных данных в искомый результат. Приведенная фраза представляет собой не определение, а объяснение сути, поскольку понятие алгоритма является фундаментальным и не может быть выражено через другие, поэтому его следует рассматривать как неопределяемое.

В качестве примера алгоритма приведем алгоритм Евклида для нахождения наибольшего общего делителя (НОД) двух натуральных чисел. Вот одна из возможных формулировок этого алгоритма, описанная по шагам:

1) присвоить переменным X и Y значения, НОД которых ищется;
2) если $X > Y$, то перейти на шаг 5;
3) если $X < Y$, то перейти на шаг 6;
4) здесь $X = Y$. Выдать X в качестве результата. Конец работы;
5) заменить пару (X, Y) парой $(X - Y, Y)$ и вернуться на шаг 2;
6) заменить пару (X, Y) парой $(X, Y - X)$ и вернуться на шаг 2.

Выяснение того, какие объекты и действия над ними следует считать точно определенными, какими возможностями обладают комбинации элементарных действий, что можно и чего нельзя сделать с их помощью, — все это предмет теории алгоритмов и формальных систем, которая первоначально возникла в рамках математики и стала важнейшей ее частью. Как указывал В. А. Успенский, самым главным открытием в науке об алгоритмах, безусловно, было открытие самого понятия алгоритма в качестве новой и отдельной сущности.

Вычисления протекают во времени и в пространстве. Каждый шаг алгоритма выполняется за какое-то конечное время. Для размещения данных необходимо пространство — память. Рассмотрим основные свойства алгоритма.

Свойства алгоритм

Каждый алгоритм имеет дело с *данными* — входными, промежуточными и выходными.

Конечность. Понимается двояко: во-первых, алгоритм состоит из отдельных элементарных шагов, или действий, причем множество различных шагов, из которых составлен алгоритм, конечно. Во-вторых, алгоритм должен заканчиваться за конечное число шагов. Если строится бесконечный, сходящийся к искомому решению процесс, то он обрывается на некотором шаге и полученное значение принимается за приближенное решение рассматриваемой задачи. Точность приближения зависит от числа шагов.

Элементарность (понятность). Каждый шаг алгоритма должен быть простым, чтобы устройство, выполняющее операции, могло выполнить его одним действием.

Дискретность. Процесс решения задачи представляется конечной последовательностью отдельных шагов, и каждый шаг алгоритма выполняется за конечное (не обязательно единичное) время.

Детерминированность (определенность). Каждый шаг алгоритма должен быть однозначно и недвусмысленно определен и не должен допускать произвольной трактовки. После каждого шага либо указывается, какой шаг делать дальше, либо дается команда остановки, после чего работа алгоритма считается законченной.

Результативность. Алгоритм имеет некоторое число входных величин — аргументов. Цель выполнения алгоритма состоит в получении конкретного результата, имеющего вполне определенное отношение к исходным данным. Алгоритм должен останавливаться после конечного числа шагов, зависящего от данных, с указанием того, что считать результатом. Если решение не может быть найдено, то должно быть указано, что в этом случае считать результатом.

Массовость. Алгоритм решения задачи разрабатывается в общем виде, т. е. он должен быть применим для некоторого класса задач, различающихся лишь исходными данными. При этом исходные данные могут выбираться из некоторой области, которая называется *областью применимости алгоритма.*

Эффективность. Одну и ту же задачу можно решить по-разному и соответственно за разное время и с различными затратами памяти. Желательно, чтобы алгоритм состоял из минимального числа шагов и при этом решение удовлетворяло бы условию точности и требовало минимальных затрат других ресурсов.

Точное математическое определение алгоритма затрудняется тем, что интерпретация предусмотренных предписаний не должна зависеть от выполняющего их субъекта. В зависимости от своего интеллектуального уровня он может либо не понять, что имеется в виду в инструкции, либо интерпретировать ее непредусмотренным образом.

Можно обойти проблему интерпретации правил, если наряду с формулировками предписаний описать конструкцию и принцип действия интерпретирующего устройства. Это позволяет из-

бежать неопределенности и неоднозначности в понимании одних и тех же инструкций. Для этого необходимо задать язык, на котором описывается множество правил поведения, либо последовательность действий, а также само устройство, которое может интерпретировать предложения, сделанные на этом языке, и выполнять шаг за шагом каждый точно определенный процесс. Оказывается, что такое устройство (машину) можно выполнить в виде, который остается постоянным независимо от сложности рассматриваемой процедуры.

В настоящее время можно выделить три основных типа универсальных алгоритмических моделей. Они различаются исходными посылками относительно определения понятия алгоритма.

Первый тип связывает понятие алгоритма с наиболее традиционными понятиями математики — вычислениями и числовыми функциями. *Второй тип* основан на представлении об алгоритме как о некотором детерминированном устройстве, способном выполнять в каждый отдельный момент лишь весьма примитивные операции. Такое представление обеспечивает однозначность алгоритма и элементарность его шагов. Кроме того, такое представление соответствует идеологии построения компьютеров. Основной теоретической моделью этого типа, созданной в 1930-х гг. английским математиком Аланом Тьюрингом, является машина Тьюринга.

Третий тип — это преобразования слов в произвольных алфавитах, в которых элементарными операциями являются подстановки, т. е. замены части слова (под словом понимается последовательность символов алфавита) другим словом. Преимущества этого типа моделей состоят в его максимальной абстрактности и возможности применить понятие алгоритма к объектам произвольной (необязательно числовой) природы. Примеры моделей третьего типа — канонические системы американского математика Эмиля Л. Поста и нормальные алгоритмы, введенные советским математиком А. А. Марковым.

Модели второго и третьего типа довольно близки и отличаются в основном эвристическими акцентами, поэтому не случайно говорят о машине Поста, хотя сам Пост такое название не вводил.

Запись алгоритма на некотором языке представляет собой программу. Если программа написана на специальном алгоритмическом языке (например, на языке ПАСКАЛЬ или БЕЙСИК), то говорят об *исходной программе.* Программа, написанная на языке, который непосредственно понимает компьютер (как правило, это двоичные коды), называется *машинной, или двоичной.*

Любой способ записи алгоритма подразумевает, что всякий описываемый с его помощью предмет задается как конкретный представитель некоторого класса объектов, которые можно описывать данным способом.

Средства, используемые для записи алгоритмов, в значительной мере определяются тем, кто будет исполнителем.

Если исполнителем будет человек, запись может быть не полностью формализована, на первое место выдвигаются понятность и наглядность. В этом случае можно использовать словесную форму записи или схемы алгоритмов.

Для записи алгоритмов, предназначенных для исполнителей-автоматов, необходима формализация, поэтому в таких случаях применяют формальные специальные языки. Преимущество формального способа записи состоит в том, что он дает возможность изучать алгоритмы как математические объекты; при этом формальное описание алгоритма служит основой, позволяющей интеллектуально охватить этот алгоритм.

Для записи алгоритмов используют самые разнообразные средства. Выбор средства определяется типом исполняемого алгоритма. Выделяют следующие **основные способы записи алгоритмов**:

— *вербальный* — алгоритм описывается на человеческом языке;

— *символьный* — алгоритм описывается с помощью набора символов;

— *графический* — алгоритм описывается с помощью набора графических изображений.

Общепринятыми способами записи алгоритма являются *графическая запись* с помощью схем алгоритмов (блок-схем) и символьная *запись* с помощью какого-либо алгоритмического языка.

Для описания алгоритма с помощью схем изображают связанную последовательность геометрических фигур, каждая из которых подразумевает выполнение определенного действия алгоритма. Порядок выполнения действий указывается стрелками.

В схемах алгоритмов используют следующие типы графических обозначений.

Начало и *конец* алгоритма обозначают с помощью одноименных символов (рис. 1.1).

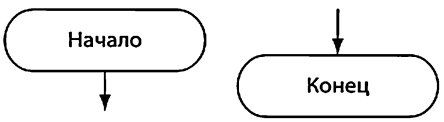

Рис. 1.1. **Обозначения начала и конца алгоритма**

Шаг алгоритма, связанный с присвоением нового значения некоторой переменной, преобразованием некоторого значения с целью получения другого значения, изображается символом «*процесс*» (рис. 1.2).

Выбор направления выполнения алгоритма в зависимости от некоторых переменных условий изображается символом «*решение*»

(рис. 1.3). Здесь *P* означает предикат (условное выражение, условие). Если условие выполнено (предикат принимает значение ИСТИНА), то выполняется переход к одному шагу алгоритма, а если не выполнено, то к другому.

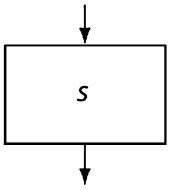

Рис. 1.2. **Обозначение процесса**

Имеются примитивы для операций ввода и вывода данных, а также другие графические символы. В настоящий момент они определены стандартом ГОСТ 19.701—90 (ИСО 5807—85) «Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения». Всего сборник ЕСПД содержит 28 документов.

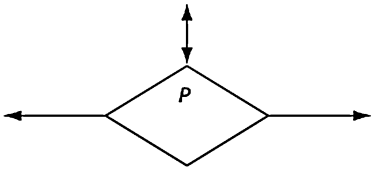

Рис. 1.3. **Обозначение решения**

По схеме алгоритма легко составить исходную программу на алгоритмическом языке.

В зависимости от последовательности выполнения действий в алгоритме выделяют алгоритмы линейной, разветвленной и циклической структуры.

В алгоритмах **линейной структуры** действия выполняются последовательно одно за другим.

В алгоритмах **разветвленной структуры** в зависимости от выполнения или невыполнения какого-либо условия производятся различные последовательности действий. Каждая такая последовательность действий называется *ветвью алгоритма.*

В алгоритмах **циклической структуры** в зависимости от выполнения или невыполнения какого-либо условия выполняется повторяющаяся последовательность действий, называющаяся *телом цикла.* Вложенным называется цикл, находящийся внутри тела другого цикла. Итерационным называется цикл, число повторений которого не задается, а определяется в ходе выполнения цикла. Одно повторение цикла называется *итерацией.*

1.2. Методы разработки алгоритмов[1]

Ключевым подходом в алгоритмизации является сведение задачи к подзадачам. Это естественно, так как предстоит превратить один шаг в последовательность элементарных шагов. Само это преобразование также может состоять из нескольких этапов, на которых единственный шаг разбивается на несколько более простых, но еще не элементарных. Эти более простые шаги соответствуют частным задачам (подзадачам), совокупное решение которых приводит к решению исходной задачи.

Различные методы разработки алгоритмов отличаются тем, на какие подзадачи производится разбиение и как эти подзадачи соотносятся между собой. Хотя общепринятой классификации методов разработки алгоритмов нет, тем не менее некоторые общие соображения на этот счет могут быть высказаны.

Любая задача может быть сформулирована как функция преобразования исходных данных в выходные данные: $f(X) = Y$. Как исходные данные, так и функция могут быть достаточно сложными. Например, X — число, вектор, текст на каком-либо языке, БД, рисунок, информация. То же можно сказать и о выходных данных. Функция может быть суммой чисел, тригонометрической функцией, корнем уравнения, переводом текста с русского на английский язык, раскраской полутонового рисунка и т. д.

Разбивая задачу на подзадачи, можно: разбивать исходные и выходные данные на части или упрощать их; производить декомпозицию функции, т. е. превращать ее в суперпозицию более простых.

Разбиение данных

Под **разбиением данных** понимается разделение структуры данных на части, например разделение вектора из десяти компонентов на два вектора по пять компонентов или разделение текста на предложения. Под упрощением данных понимаются такие ситуации, когда, например, X — число и его нельзя разбить на части, но его можно разложить, скажем, на сумму $X = X_1 + X_2$ так, что результаты $f(X_1), f(X_2)$ отыскиваются проще, чем $f(X)$.

При разработке алгоритма может использоваться отдельно первый подход, отдельно второй подход или оба подхода совместно.

Разложение задачи на подзадачи

Разложение задачи в **последовательность разнородных подзадач** иногда называют методом «разделяй и властвуй».

В этом методе обычно выделяется относительно небольшое число подзадач. Например: задача — выполнить программу на компьютере; подзадачи — ввести исходный текст программы; транслировать

[1] Использованы материалы сайта http://it.kgsu.ru (автор — Т. А. Никифорова).

программу в машинные команды; подсоединить к машинному коду стандартные процедуры из библиотеки; загрузить программу в оперативную память; запустить процесс выполнения; завершить процесс выполнения программы.

Результаты решения первой подзадачи становятся исходными данными для второй подзадачи и т. д. Таким образом, здесь использован второй подход в чистом виде (декомпозиция функции, задание ее в виде суперпозиции более простых). Заметим также, что такая суперпозиция может быть задана последовательным соединением машин Тьюринга. На алгоритмическом языке этот метод может быть выражен записью последовательно вызываемых процедур.

Важный частный случай предыдущего метода — разложение задачи в **последовательность однородных подзадач** приобретает новое качество за счет того, что задача P сводится к n экземплярам более простой задачи R и к простой задаче Q, объединяющей n решений.

Пример

Вычислим скалярное произведение двух векторов — A и B:

```
S: = 0;          {Задача Q — подготовка места для суммирования}
for i: = 1 to n do
S: = S+A[i]*B[i];  {Задача R — перемножение компонент
                    и суммирование}
```

Этот алгоритм использует разбиение исходных данных на части — отдельные компоненты векторов.

Однородность подзадач позволяет значительно сократить длину текста алгоритма за счет применения операторов повторения. Итерация на уровне крупных подзадач или отдельных небольших операторов встречается в большинстве реальных алгоритмов и служит основным источником эффективного использования компьютера по сравнению с другими вычислительными средствами (например, непрограммируемым калькулятором).

Рекурсия

Рекурсия — это сведение задачи к самой себе. Задача так же, как и в предыдущем методе, сводится к более простой. Но эта более простая задача имеет ту же формулировку, что и исходная, с той лишь разницей, что решаться она должна для более простых исходных данных. Это чистый вариант упрощения исходных данных.

Метод последовательных приближений

Сначала каким-либо образом угадывается значение x_0, близкое к решению. Задача P нахождения решения сводится к многократ-

ному решению задачи R улучшения решения. Метод предполагает, что каким-то образом может быть оценено «качество» решения (обычно — точность). Чаще всего абсолютная точность недостижима, поэтому процесс потенциально бесконечен, т. е. не выполняется свойство конечности алгоритма. Для того чтобы этого избежать, несколько изменяют первоначальную формулировку задачи: требуют отыскать не точное решение Y, а любое решение, отличающееся от Y не более чем на некоторую величину e, т. е. приближенное решение. Характерный пример — задача отыскания корня уравнения или задача отыскания корня p-й степени из x.

Основной проблемой является построение задачи R по исходной задаче P, доказательство факта сходимости процесса к искомому решению и обеспечение достаточно высокой скорости сходимости.

Решение обратной задачи

Иногда обратная задача т. е. задача соответствующая функции $f'(Y) = X$, решается значительно более просто, чем исходная задача. Тогда имеющийся алгоритм решения обратной задачи R иногда можно использовать для построения алгоритма решения прямой задачи P.

Метод полного перебора

Берется некоторое подмножество исходных данных и непосредственной проверкой (решением задачи) проверяется, удовлетворяет ли это подмножество поставленному условию. Поскольку различных подмножеств имеется конечное число, то потенциально можно перебрать все подмножества и найти решение.

Сложность заключается в том, что с увеличением количества исходных данных (переходе от k к $k + 1$) быстро увеличивается необходимое число проверок. Метод полного перебора применим в тех случаях, когда искомое решение $Y = f(X)$ принадлежит некоторой конечной области и может быть найдена простая функция **quality**(Y) для проверки правильности (или качества) выбранного решения. Тогда задача P вычисления функции f заменяется многократным решением задачи R вычисления функции **quality** (столько раз, сколько элементов имеется в области решений). Причем в общем случае просмотреть нужно всю область, и порядок, в котором просматриваются элементы, не важен.

Эвристические методы разработки алгоритмов

Под *эвристическими* понимаются методы, правильность которых не доказана. Они выглядят правдоподобными, кажется, что в большинстве случаев они должны давать верное решение. Иногда не удается построить контрпример, демонстрирующий ошибочность или неуниверсальность метода. Но не удается доказать ма-

тематическими средствами и правильность метода. Тем не менее практика использования эвристических методов дает положительные результаты.

Динамическое программирование

В наиболее общей форме так называют процесс пошагового решения задач, когда на каждом шаге выбирается одно значение из множества допустимых на этом шаге, причем такое, которое оптимизирует заданную цель.

Часто не удается разбить задачу на небольшое число подзадач, объединенное решение которых позволяет получить решение искомых задач. Можно попытаться разделить задачу на столько задач, сколько необходимо, затем каждую на еще более мелкие и т. д. Иногда проще создать таблицу решения всех подзадач, независимо от того, нужна она или нет. Заполнение таблиц — решение подзадач для получения решения определенной задачи — в теории алгоритмов получило название *динамического программирования*. Формы алгоритмов динамического программирования могут быть разными, общими могут быть лишь заполненные таблицы и порядок заполнения их элементов.

Тема 2
ОСНОВНЫЕ ПОНЯТИЯ ЯЗЫКА ВЫСОКОГО УРОВНЯ

2.1. Эволюция и классификация языков программирования[1]

Для самых первых компьютеров, появившихся в 1940—1950-х гг., программирование велось непосредственно в машинных кодах, а основным носителем информации были перфокарты и перфоленты. Программисты обязаны были досконально знать архитектуру машины. Программы были достаточно простыми, что обусловливалось, во-первых, весьма ограниченными возможностями этих машин и, во-вторых, большой сложностью разработки и, главное, отладки программ непосредственно на машинном языке.

АССЕМБЛЕР

Первым этапом развития языков программирования является появление языка АССЕМБЛЕР, который обеспечивал возможность символического кодирования машинных команд, т. е. обозначения их с помощью осмысленных названий. Одна команда АССЕМБЛЕРа может представлять собой не одну машинную команду, а целую последовательность.

Программисту больше не требовалось вникать в хитроумные способы кодирования команд на аппаратном уровне. Более того, зачастую одинаковые по сути команды кодировались различным образом в зависимости от своих параметров. Появилось даже некое подобие *переносимости* — существовала возможность разработки целого семейства машин со сходной системой команд и некоего общего ассемблера для них, при этом не было нужды обеспечивать двоичную совместимость.

ФОРТРАН

В 1954 г. группой разработчиков во главе с Джоном Бэкусом был создан язык программирования ФОРТРАН (FORTRAN — от **For**mula

[1] Использованы материалы из статьи М. Плискина «Эволюция языков программирования», URL: http://schools.keldysh.ru/sch444/MUSEUM/LANR/evol.htm, http://www. dotsite.ru/Publications/Publication51.aspx.

Translator, т. е. язык для программирования математических расчетов).

Это первый язык программирования высокого уровня. Впервые программист мог по-настоящему абстрагироваться от особенностей машинной архитектуры. Ключевой идеей, отличающей новый язык от АССЕМБЛЕРа, была концепция подпрограмм. Транслятор с ФОРТРАНа в машинные коды был весьма сложной программой, которая в отличие от современных трансляторов не выполняла контроль многих ошибок программиста. Кроме того, генерируемый машинный код был менее эффективен, чем если бы программа изначально писалась на АССЕМБЛЕРе.

Со временем пришло понимание того, что реализация больших проектов невозможна без применения языков высокого уровня. Мощность вычислительных машин росла, и с тем падением эффективности, которое раньше считалось угрожающим, стало возможным смириться. Преимущества же языков высокого уровня стали настолько очевидными, что побудили разработчиков к созданию новых языков, все более и более совершенных.

ФОРТРАН был задуман для использования в сфере научных и инженерно-технических вычислений. На этом языке легко описываются задачи с разветвленной логикой (моделирование производственных процессов, решение игровых ситуаций и т. д.), некоторые экономические задачи и особенно задачи редактирования (составление таблиц, сводок, ведомостей и т. д.).

За длительное время использования этого языка-долгожителя были накоплены колоссальные библиотеки стандартных подпрограмм, реализующих вычисления различного рода, таких как, например, численные методы решения дифференциальных уравнений. В настоящее время применяются современные версии ФОРТРАНа, возможности которых близки к другим языкам высокого уровня.

КОБОЛ

В 1960 г. был создан язык программирования КОБОЛ (COBOL — *Common Business-Oriented Language*). Он задумывался как язык для создания коммерческих приложений. На КОБОЛе написаны тысячи прикладных коммерческих систем. Отличительной особенностью языка является возможность эффективной работы с большими массивами данных. Популярность КОБОЛа столь высока, что даже сейчас, при всех его недостатках, появляются его новые диалекты и реализации. Например, недавно появилась реализация КОБОЛа для Microsoft.NET.

PL/1

В 1964 г. в корпорации IBM был создан язык ПЛ/1 (PL/1 — Programming Language 1), который был призван заменить КОБОЛ

и ФОРТРАН в большинстве приложений. Язык обладал исключительным богатством синтаксических конструкций. Однако язык оказался крайне сложным для написания и в особенности отладки программ, поэтому так и не приобрел широкую популярность.

БЕЙСИК

В 1963 г. в Дартмутском колледже был разработан язык программирования БЕЙСИК (BASIC — **B**eginners' **A**ll-Purpose **S**ymbolic **I**nstruction **C**ode — многоцелевой язык символических инструкций для начинающих). Язык задумывался в первую очередь как средство первоначального обучения программированию. БЕЙСИК действительно стал языком, на котором учились программировать несколько поколений. БЕЙСИК был также доступен на ПК, обычно он встроен в ПЗУ. Так БЕЙСИК завоевал популярность. Он и сегодня самый простой для освоения из десятков языков общецелевого программирования. Было создано несколько мощных реализаций БЕЙСИКа, поддерживающих самые современные концепции программирования (ярчайший пример — Microsoft Visual Basic).

АЛГОЛ

В 1960 г. командой во главе с Петером Науром был создан язык программирования АЛГОЛ (ALGOL — **Algo**rithmic **L**anguage). Этот язык дал начало целому семейству алголоподобных языков (важнейший представитель — ПАСКАЛЬ). В 1968 г. появилась новая версия языка. Она не нашла столь широкого практического применения, как первая версия, но была весьма популярна в кругах теоретиков. Язык обладал многими уникальными на тот момент характеристиками.

Эти языки (ФОРТРАН, КОБОЛ, ПЛ/1, БЕЙСИК, АЛГОЛ и др.) послужили фундаментом для более поздних разработок. Все они представляют одну и ту же парадигму программирования — императивную[1].

ПАСКАЛЬ

В 1970 г. Никлаусом Виртом был создан язык программирования ПАСКАЛЬ (PASCAL, в честь ученого Блеза Паскаля). Язык замечателен тем, что это первый широко распространенный язык для *структурного программирования*. Впервые оператор безусловного перехода перестал играть основополагающую роль при управлении порядком выполнения операторов. В этом языке также внедрена строгая проверка типов, что позволило выявлять многие ошибки на этапе компиляции.

Хотя ПАСКАЛЬ был разработан как язык для обучения программированию, он впоследствии получил широкое развитие и в настоящее время считается одним из самых используемых языков. Отрицательной чертой языка было отсутствие в нем средств для разбиения

[1] О парадигмах программирования рассказывается в теме 6.

программы на модули. Никлаус Вирт это осознавал и разработал язык Modula-2 (1978), в котором идея модуля стала одной из ключевых концепций языка. В 1988 г. появилась Modula-3, в которую были добавлены объектно-ориентированные черты. Логическим продолжением языков ПАСКАЛЬ и Modula являются языки Oberon и Oberon-2. Они характеризуются движением в сторону объектно- и компонентно-ориентированности.

С-подобные языки

В 1972 г. Брайан Керниган и Деннис Ритчи создали *язык программирования С* (Си) (это была третья, удачная, версия, первой и второй были соответственно языки *A* и *B*). Язык С создавался как язык для разработки операционной системы UNIX. Его часто называют «переносимым ассемблером», поскольку он позволяет работать с данными практически так же эффективно, как на АССЕМБЛЕРе, предоставляя при этом структурированные управляющие конструкции и абстракции высокого уровня (структуры и массивы). Именно с этим связана его огромная популярность и поныне. И именно это является его ахиллесовой пятой. Компилятор языка Си очень слабо контролирует типы, поэтому очень легко написать внешне совершенно правильную, но логически ошибочную программу.

В 1986 г. Бьярн Страуструп создал первую версию *языка С++*, добавив в язык С объектно-ориентированные черты, взятые из Simula (см. далее), и исправив некоторые ошибки и неудачные решения языка. С++ продолжает совершенствоваться и в настоящее время. Так, в 1998 г. вышла новая (третья) версия стандарта. Язык стал основой для разработки современных больших и сложных проектов. Однако у него имеются и слабые стороны, связанные с его сложностью и неэффективностью некоторых аспектов.

В 1995 г. в корпорации Sun Microsystems Кеном Арнольдом и Джеймсом Гослингом был создан язык *Java* (назван по марке кофе). Он наследовал синтаксис С и С++ и был избавлен от некоторых неприятных черт последнего. Отличительной особенностью языка является компиляция в код некоей абстрактной машины, для которой затем пишется эмулятор (*Java Virtual Machine*) для реальных систем (см. далее).

В 1999—2000 гг. в корпорации Microsoft был создан язык *C#*. Он в достаточной степени схож с Java (и задумывался как альтернатива последнему), но имеет и отличительные особенности. Ориентирован в основном на разработку многокомпонентных интернет-приложений.

Языки Ada и Ada-95

В 1983 г. под эгидой Министерства Обороны США был создан язык Ada (в честь первой программистки Ады Лавлейс). Язык за-

мечателен тем, что очень много ошибок может быть выявлено на этапе компиляции. Кроме того, поддерживаются многие аспекты программирования, которые часто отдаются на откуп ОС (параллелизм, обработка исключений). В 1995 г. был принят стандарт языка Ada-95, который развивает предыдущую версию, добавляя в нее объектную ориентированность и исправляя некоторые неточности. Оба этих языка не получили широкого распространения вне военных и прочих крупномасштабных проектов (авиация, железнодорожные перевозки). Основной причиной является сложность освоения языка и достаточно громоздкий синтаксис.

Языки обработки данных

Все перечисленные ранее языки являются языками общего назначения в том смысле, что они не ориентированы и не оптимизированы для использования каких-либо специфических структур данных или для применения в каких-либо специфических областях. Было разработано большое количество языков, ориентированных на достаточно специфические применения. Приведем несколько примеров таких языков.

В 1957 г. была предпринята попытка создания языка для описания математической обработки данных. Язык был назван APL (*Application Programming Language*). Его отличительной особенностью было использование математических символов и очень мощный синтаксис, который позволял производить множество нетривиальных операций прямо над сложными объектами, не прибегая к разбиению их на компоненты. Широкому применению помешало использование нестандартных символов как элементов синтаксиса.

В 1962 г. появился язык *Snobol* (а в 1974 г. — его преемник *Icon*), предназначенный для обработки строк. Синтаксис *Icon* напоминает Си и ПАСКАЛЬ одновременно. Отличие заключается в наличии мощных встроенных функций работы со строками и связанная с этими функциями особая семантика. Современным аналогом Icon и Snobol является Perl — язык обработки строк и текстов, в который добавлены некоторые объектно-ориентированные возможности.

Скриптовые языки

В последнее время в связи развитием интернет-технологий, широким использованием высокопроизводительных компьютеров и рядом других факторов получили распространение так называемые *скриптовые языки*. Эти языки первоначально ориентировались на использование в качестве внутренних управляющих языков во всякого рода сложных системах.

Многие из них, однако, вышли за пределы сферы своего изначального применения и используются ныне в совсем иных областях. Характерными особенностями данных языков являются, во-

первых, их интерпретируемость (компиляция либо невозможна, либо нежелательна[1]), во-вторых, простота синтаксиса, а в-третьих, легкая расширяемость. Таким образом, они идеально подходят для использования в часто изменяемых программах, очень небольших программах или в случаях, когда для выполнения операторов языка затрачивается время, несопоставимое со временем их разбора. Было создано достаточно большое количество таких языков, перечислим лишь основные и наиболее часто используемые.

JavaScript

Язык был создан в компании Netscape Communications в качестве языка для описания сложного поведения веб-страниц. Интерпретируется браузером во время отображения веб-страницы. По синтаксису схож с Java и отдаленно с C/C++. Имеет возможность использовать встроенную в браузер объектную функциональность, однако подлинно объектно-ориентированным языком не является.

VBScript

Язык был создан в корпорации Microsoft во многом в качестве альтернативы JavaScript. Имеет схожую область применения. Синтаксически схож с языком Visual Basic и является его усеченной версией. Так же, как и JavaScript, исполняется браузером при отображении веб-страниц и имеет ту же степень объектной ориентированности.

Perl

Язык создавался в помощь системному администратору ОС Unix для обработки различного рода текстов и выделения нужной информации. Развился до мощного средства работы с текстами. Является интерпретируемым языком и реализован практически на всех существующих платформах. Применяется при обработке текстов, а также для динамической генерации веб-страниц на веб-серверах.

Python

Интерпретируемый объектно-ориентированный язык программирования.

По структуре и области применения близок к Perl, однако более строг и логичен. Имеются реализации для большинства существующих платформ.

Объектно-ориентированные языки

Объектно-ориентированный подход, пришедший на смену структурному, впервые появился в языке СИМУЛА (Simula, 1967). Этот язык был предназначен для моделирования различных объектов

[1] О компиляции и интерпретации рассказывается в следующем разделе.

и процессов, и объектно-ориентированные черты появились в нем именно для описания свойств модельных объектов.

Популярность объектно-ориентированному программированию (ООП) принес *язык Smalltalk*, созданный в 1972 г. Язык предназначался для проектирования сложных графических интерфейсов и был первым по-настоящему объектно-ориентированным языком. В нем классы и объекты — единственные конструкции программирования. Большим *недостатком* Smalltalk являются большие требования к памяти и низкая производительность полученных программ.

Существует язык с очень хорошей реализацией объектно-ориентированности, не являющийся надстройкой ни над каким другим языком. Это *язык Eiffel* (1986). В настоящее время существует реализация Eiffel для Microsoft.NET.

Языки параллельного программирования

Большинство компьютерных архитектур и языков программирования ориентированы на последовательное выполнение операторов программы. В настоящее время существуют программно-аппаратные комплексы, позволяющие организовать параллельное выполнение различных частей одного и того же вычислительного процесса. Для программирования таких систем необходима специальная поддержка со стороны средств программирования, в частности языков программирования. Некоторые языки общего назначения содержат в себе элементы поддержки параллелизма, однако программирование истинно параллельных систем требует подчас специальных приемов.

Язык *Occam* был создан в 1982 г. и предназначен для программирования транспьютеров — многопроцессорных систем распределенной обработки данных. Он описывает взаимодействие параллельных процессов в виде каналов — способов передачи информации от одного процесса к другому.

В 1985 г. была предложена модель параллельных вычислений *Linda*. Основной ее задачей является организация взаимодействия между параллельно выполняющимися процессами. Linda может быть добавлена в любой язык программирования. Существуют достаточно эффективные реализации Linda.

Неимперативные языки

Все языки, о которых шла речь ранее, имеют одно общее свойство: они реализуют императивную парадигму. Это означает, что программы на них в конечном итоге представляют собой пошаговое описание решения той или иной задачи. Можно попытаться описывать лишь постановку проблемы, а решать задачу поручить компилятору. Существует два основных подхода, развивающие эту идею: функциональное и логическое программирование.

Функциональные языки

Основная идея, лежащая в основе функционального программирования, — это представление программы в виде математических функций (т. е. функций, значение которых определяется лишь их аргументами, а не контекстом выполнения). Оператор присваивания в таких языках не используется (или, как минимум, его использование не поощряется). Императивные возможности, как правило, имеются, но их применение обставлено серьезными ограничениями.

В конце 1950-х гг. появился язык ЛИСП (LISP — List Information Symbol Processing, язык для обработки списков) для работы со строками символов. Это особое предназначение ЛИСПа открыло для программистов новую область деятельности, известную ныне как «искусственный интеллект». В настоящее время ЛИСП успешно применяется в экспертных системах, системах аналитических вычислений и т. п.

Обширность области возможных приложений ЛИСПа вызвала появление множества различных диалектов ЛИСПа. Это легко объяснимо: применение ЛИСПа для понимания естественного языка требует определенного набора базисных функций, отличных, например, от используемого в задачах медицинской диагностики. Потомки ЛИСП — Planner (1967), Scheme (1975), Common Lisp (1984). Последний был принят в университетах США, а также многими разработчиками систем искусственного интеллекта в качестве основного диалекта языка ЛИСП. Многие его черты были унаследованы современными языками функционального программирования.

Кроме перечисленных, можно упомянуть ML и два его современных диалекта — Standard ML (SML) и CaML. Последний имеет объектно-ориентированного потомка — Objective CaML (O'CaML), а также Haskell и его более простой диалект Clean.

Языки логического программирования

Согласно логическому подходу к программированию, программа представляет собой совокупность правил, или логических высказываний. Кроме того, в программе допустимы логические причинно-следственные связи. Таким образом, языки логического программирования базируются на классической логике и применимы для систем логического вывода, в частности, для так называемых *экспертных систем*. На языках логического программирования естественно формализуется логика поведения, и они применимы для описаний правил принятия решений, например, в системах, ориентированных на поддержку бизнеса.

Важным преимуществом такого подхода является достаточно высокий уровень машинной независимости, а также возможность *откатов* — возвращения к предыдущей подцели при отрицательном результате анализа одного из вариантов в процессе поиска реше-

ния (скажем, очередного хода при игре в шахматы), что избавляет от необходимости поиска решения путем полного перебора вариантов и увеличивает эффективность реализации.

Программы на языках логического программирования выражены как формулы математической логики, а компилятор пытается получить следствия из них. Родоначальником большинства языков логического программирования является язык ПРОЛОГ (Prolog, 1971). У него есть ряд потомков — Parlog (1983, ориентирован на параллельные вычисления), Delta Prolog и др.

ПРОЛОГ — это язык, предназначенный для поиска решений. Это декларативный язык, т. е. формальная постановка задачи может быть использована для ее решения. ПРОЛОГ определяет логические отношения в задаче как отличные от пошагового решения этой задачи.

Центральной частью ПРОЛОГа являются средства логического вывода, которые решают запросы, используя заданное множество фактов и правил, к которым обращаются как к утверждениям. ПРОЛОГ также не имеет деления переменных на типы и может динамически добавлять правила и факты к средствам вывода.

Для удобства восприятия приведем таблицу с характеристиками основных языков программирования (табл. 2.1).

Таблица 2.1

Характеристика основных языков программирования

Язык	Основное использование	Описание
Ada	В оборонной промышленности	Высокого уровня
АССЕМБЛЕР	Для работ, требующих детального контроля за аппаратным обеспечением, быстрого исполнения и программ малого размера	Быстрый и эффективный, но требующий определенных усилий и навыков
БЕЙСИК	В образовании, бизнесе, дома	Прост в изучении
C	В системном, универсальном программировании	Быстрый и эффективный, широко используется как универсальный язык
C++	В объектно-ориентированном программировании	Основан на языке C
КОБОЛ	В программировании в бизнесе	Жестко ориентирован на коммерческие задачи, легко научиться, но очень много операторов
ФОРТРАН	Для научной работы и вычислений	Основан на математических формулах
ЛИСП	Для искусственного интеллекта	Язык символов с репутацией трудно изучаемого

Окончание табл. 2.1

Язык	Основное использование	Описание
МОДУЛА-2	В системном программировании, программировании в режиме реального времени и универсальном программировании	Высоко структурирован, разработан взамен языка ПАСКАЛЬ
ОБЕРОН	В универсальном программировании	Небольшой, компактный язык, соединяющий многие черты языков ПАСКАЛЬ и МОДУЛА-2
ПАСКАЛЬ	Универсальный язык	Высоко структурирован
ПРОЛОГ	Для искусственного интеллекта	Символьно-логическая система программирования, в начале предназначенная для решения теорем, но сейчас использующаяся чаще для решения задач, связанных с искусственным интеллектом

Классификация языков программирования

Существует множество классификаций языков программирования по различным критериям. Самое простое деление — на языки высокого и низкого уровня.

Язык низкого уровня — это язык программирования, предназначенный для определенного типа компьютера и отражающий его внутренний машинный код; языки низкого уровня часто называют *машинно-ориентированными языками*. Их сложно конвертировать для использования на компьютерах с разными центральными процессорами, а также довольно сложно изучать, поскольку для этого требуется хорошо знать внутренние принципы работы компьютера.

Язык высокого уровня — это язык программирования, предназначенный для удовлетворения требований программиста; он не зависит от внутренних машинных кодов компьютера любого типа. Языки высокого уровня используют для решения проблем, и поэтому их часто называют *проблемно-ориентированными языками*. Каждая команда языка высокого уровня эквивалентна нескольким командам в машинных кодах, поэтому программы, написанные на языках высокого уровня, более компактны, чем аналогичные программы в машинных кодах.

Другая классификация делит языки на *вычислительные* и языки *символьной обработки*. К первому типу относят ФОРТРАН, ПАСКАЛЬ, АЛГОЛ, БЕЙСИК, С, ко второму — ЛИСП, ПРОЛОГ, СНОБОЛ и др.

Классификация языков программирования по *типам задач* приведена в табл. 2.2.

Таблица 2.2

Классификация языков программирования по типам задач

Тип задачи	Языки программирования
Задачи искусственного интеллекта	ЛИСП, ПРОЛОГ, Common Lisp, РЕФАЛ, Planner, QLisp
Параллельные вычисления	Fun, Apl, ML, SML, Occam, Actus, параллельный КОБОЛ, ОВС-АЛГОЛ, ОВС-ФОРТРАН
Задачи вычислительной математики и физики	Occam, Actus, параллельный КОБОЛ, ОВС-АЛГОЛ, ОВС-ФОРТРАН
Разработка интерфейса, программ-оболочек, систем	Forth, C, C++, АССЕМБЛЕР, МАКРОАССЕМБЛЕР, СИМУЛА-67, ОАК, Smalltalk, Java, РПГ
Задачи вычислительного характера	АЛГОЛ, ФОРТРАН, КОБОЛ, Ada, PL/1, БЕЙСИК, ПАСКАЛЬ
Оформление документов, обработка больших текстовых файлов, организация виртуальных трехмерных интерфейсов в Интернете, разработка БД	HTML, Perl, SQL, Informix 4GL, Natural, DDL, DSDL, SEQUEL

Еще одна распространенная классификация языков программирования основана на принципе их организации, или *парадигме* (подробнее см. тему 6). По этой классификации языки делят на *процедурные* (употребляются также термины *императивные* и *структурные*, хотя это не совсем одно и то же), *объектно-ориентированные*, *функциональные* и *логические*.

В **процедурных языках** программа явно описывает действия, которые необходимо выполнить, а результат задается только способом получения его при помощи некоторой процедуры, которая представляет собой определенную последовательность действий. В эту большую группу входят, например, ПАСКАЛЬ, С, АДА, ПЛ/1, ФОРТРАН и БЕЙСИК.

В **объектно-ориентированных языках** не описывают подробной последовательности действий для решения задачи, хотя они содержат элементы процедурного программирования. Программа пишется в терминах объектов, которые обладают свойствами и поведением. Объекты обмениваются сообщениями.

В **функциональных языках** программа описывает вычисление некоторой функции. Обычно эта функция задается как композиция других, более простых, те в свою очередь разлагаются на еще более простые и т. д. Один из основных элементов в функциональных языках — рекурсия, т. е. вычисление значения функции через значе-

ние этой же функции от других элементов. Присваивания и циклов в классических функциональных языках нет. Представителями этой группы являются ЛИСП, ML и Haskell.

В **логических языках** программа вообще не описывает действий. Она задает данные и соотношения между ними. После этого системе можно задавать вопросы. Машина перебирает известные и заданные в программе данные и находит ответ на вопрос. Порядок перебора не описывается в программе, а неявно задается самим языком. Классическим языком логического программирования считается ПРОЛОГ. Построение логической программы вообще не требует алгоритмического мышления, программа описывает статические отношения объектов, а динамика находится в механизме перебора и скрыта от программиста.

Функциональные и логические языки называют *декларативными*, или *непроцедурными*, поскольку программа представляет собой не набор команд, а описание действий, которые необходимо осуществить. Этот подход существенно проще и прозрачнее формализуется математическими средствами. Следовательно, программы проще проверять на наличие ошибок (тестировать), а также на соответствие заданной технической спецификации (верифицировать). Высокая степень абстракции также является преимуществом данного подхода. Фактически программист оперирует не набором инструкций, а абстрактными понятиями, которые могут быть достаточно обобщенными.

На начальном этапе развития декларативным языкам программирования было сложно конкурировать с императивными в силу объективных трудностей эффективной реализации трансляторов. Программы работали медленнее, однако они могли решать более абстрактные задачи с меньшими трудозатратами.

2.2. Программа, порядок ее разработки и исполнения

Для решения задачи на компьютере требуется написать программу. *Программа* на языке высокого уровня состоит из исполняемых операторов и операторов описания.

Исполняемый оператор задает законченное действие, выполняемое над данными. Примеры операторов: вывод на экран, занесение числа в память, выход из программы.

Оператор описания, как и следует из его названия, описывает данные, над которыми в программе выполняются действия. Примером описания (конечно, не на языке ПАСКАЛЬ, а на естественном языке) может служить предложение «В памяти следует отвести место для хранения целого числа, и это место мы будем обозначать А».

Описания должны предшествовать операторам, в которых используются соответствующие данные. Операторы программы исполняются последовательно, один за другим, если явным образом не задан иной порядок.

Чтобы лучше представлять себе, о чем идет речь, рассмотрим простейшую программу на языке ПАСКАЛЬ. Все, что она делает, это вычисляет и выводит на экран сумму двух целых чисел, введенных с клавиатуры:

```
var a, b, sum : integer;                                    {1}
begin                                                       {2}
readln(a, b);                                               {3}
sum := a + b;                                               {4}
writeln( 'Сумма чисел ', a, ' и ' b, ' равна ', sum );      {5}
end                                                         {6}
```

В программе шесть строк, каждая из них для удобства рассмотрения помечена комментарием с номером.

В первой строке располагается оператор описания величин, которые будут использоваться в программе. Для каждой величины задается имя, по которому к ней будут обращаться, и ее тип. «Волшебным» словом var обозначается тот факт, что a, b и sum — *переменные*, т. е. величины, которые во время работы программы могут изменять свои значения. Для всех переменных задан целый тип, он обозначается integer. Тип необходим, для того чтобы переменным в памяти было отведено соответствующее место.

Исполняемые операторы программы располагаются между служебными словами begin и end, которые предназначены для объединения операторов и сами операторами не являются. Операторы отделяются друг от друга точкой с запятой.

Ввод с клавиатуры выполняется в третьей строке с помощью стандартной процедуры с именем readln. В скобках после имени указывается, каким именно переменным будут присвоены значения. Для вывода результатов работы программы в пятой строке используется стандартная процедура writeln. В скобках через запятую перечисляется все, что надо вывести на экран, при этом пояснительный текст заключается в апострофы. Например, если ввести в программу числа 2 и 3, результат будет выглядеть так:

```
Сумма чисел 2 и 3 равна 5
```

В четвертой строке выполняется вычисление суммы и присваивание ее значения переменной sum. Справа от знака операции присваивания, обозначаемой символами :=, находится так называемое выражение. *Выражение* — это правило вычисления значения. Выражения являются частью операторов.

Компиляция и интерпретация. Исходный и объектный модули, исполняемая программа и библиотеки (модули) программ

Чтобы выполнить программу, написанную на языке программирования высокого уровня, требуется перевести ее на язык, понятный процессору — в машинные коды. Этим занимаются специальные программы, которые называются *языковыми процессорами*, или *трансляторами*. Различают два вида языковых процессоров: интерпретаторы и компиляторы.

Интерпретатор — это программа, которая получает исходную программу на языке высокого уровня и по мере распознавания его операторов выполняет описываемые ими действия.

Интерпретатор в течение всего времени работы программы находится в оперативной памяти компьютера наряду с исходным текстом программы. Интерпретатор считывает ее первый оператор, переводит его в машинные команды и тут же организует выполнение этих команд. Затем переходит к переводу и выполнению следующего оператора, и так до конца программы. При этом результаты предыдущих переводов в памяти не сохраняются. При повторном выполнении одного и того же оператора в цикле оператор снова будет переводиться в машинные коды, перед этим происходит его синтаксический анализ.

Трансляция (от лат. *translatio* — передача) — в соответствии с ГОСТ 19781—90 преобразование программы, представленной на одном языке программирования, в программу на другом языке программирования, в определенном смысле равносильную первой.

Транслятор — это программа, которая получает на вход исходную программу и формирует на выходе объектную программу (программу на объектном языке программирования). В частном случае объектным кодом может служить машинный язык, и в этом случае полученную на выходе транслятора программу можно сразу же выполнить на компьютере.

В общем случае объектный язык необязательно должен быть машинным или близким к нему (автокодом). В качестве объектного языка может служить и некоторый промежуточный язык. Для промежуточного языка может быть использован другой компилятор или интерпретатор — с промежуточного языка на машинный. Схему трансляции, когда исходная программа переводится на промежуточный язык, который затем интерпретируется, называют *гибридной*. Такая схема используется в языках Java и C#.

Транслятор, использующий в качестве входного языка близкий к машинному (автокод или язык АССЕМБЛЕР), традиционно называют *ассемблером*.

Компилятор работает следующим образом. Получив на вход исходный текст программы, компилятор выделяет из него лексемы, а затем на основе грамматики языка распознает выражения и операторы, по-

строенные из этих лексем. При этом компилятор выявляет синтаксические ошибки и в случае их отсутствия строит *объектный модуль*.

Каждый оператор языка переводится компилятором в последовательность машинных команд, которая может быть весьма длинной, поэтому языки типа ПАСКАЛЬ и С++ и называются языками высокого уровня. В языках низкого уровня, например в АССЕМБЛЕРЕ, каждая команда переводится в одну или несколько машинных команд.

Кроме того, компилятор планирует размещение данных в оперативной памяти в соответствии с описаниями величин, используемых в программе. Попутно он ищет синтаксические ошибки, т. е. ошибки записи операторов. В языке ПАСКАЛЬ на компилятор возложена еще одна обязанность — подключение к программе стандартных подпрограмм (например, ввода данных или вычисления синуса угла), а в С-подобных языках эту функцию выполняет специальная программа, называемая *компоновщиком*, или *редактором связей*.

Компоновщик, или **редактор связей**, формирует исполняемый модуль программы, подключая к объектному модулю другие объектные модули, в том числе содержащие функции библиотек, обращение к которым содержится в любой программе (например, для осуществления вывода на экран). Если программа состоит из нескольких исходных файлов, они компилируются по отдельности и объединяются на этапе компоновки. Исполняемый модуль имеет расширение *.exe* и запускается на выполнение обычным образом, при этом присутствия компилятора в памяти компьютера не требуется.

Таким образом, при компиляции трансляция и исполнение программы идут последовательно друг за другом. При интерпретации — параллельно. Один раз откомпилированная программа может быть сохранена во внешней памяти и затем исполняться многократно. На компиляцию машинное время тратиться больше не будет. Программа на интерпретируемом языке при каждом выполнении подвергается повторной трансляции. Кроме того, интерпретатор может занимать значительное место в оперативной памяти.

В самом языке программирования, вообще говоря, не заложен способ его реализации, однако одни языки почти всегда компилируются, например С++, другие, например Smalltalk, почти всегда интерпретируются; Java компилируется в байт-код и затем интерпретируется.

В настоящее время практически любая реализация языка представлена как среда разработки, которая включает в себя:

1) компилятор (или интерпретатор);
2) отладчик — специальную программу, которая облегчает процесс поиска ошибок; пользуясь ею, разработчик может выполнять программу «по шагам», отслеживать изменение значений переменных в процессе выполнения и др.;
3) встроенный текстовый редактор;

4) специальные средства для просмотра структуры программы;
5) библиотеку готовых модулей, классов, например для создания пользовательского интерфейса (окна, кнопки и т. д.).

В 1980-е гг. активно прорабатывалась идея визуального программирования, основной смысл которой состоит в том, чтобы процесс «сборки» программы осуществлялся на экране дисплея из программных конструкций-картинок. В результате появились среды разработки четвертого поколения (4GL), в которых разрабатываемый программный продукт строится из готовых крупных блоков при помощи мыши. Примерами таких сред являются: Delphi, Visual Java, Microsoft Visual Studio.NET.

Порядок разработки программы

С помощью компьютера можно решать задачи различного характера, например научно-инженерные, задачи разработки системного программного обеспечения, программ для обучения, для управления производственными процессами и т. д.

Разные задачи требуют различных технологий своей разработки, различающиеся принципами, количеством и составом этапов, областями применения и назначением. Для создания относительно простых программ успешно применяется *классический жизненный цикл разработки*, когда очередной этап начинается после полного завершения предыдущего. Обычно выделяют следующие этапы:

1) постановка задачи;
2) математическое описание задачи;
3) выбор и обоснование метода решения;
4) выбор структур данных и алгоритмов решения задачи;
5) составление (кодирование) программы;
6) тестирование и отладка программы;
7) анализ результатов.

Некоторые этапы могут отсутствовать, например в задачах разработки системного программного обеспечения отсутствует математическое описание.

Подробнее об этом рассказывается в теме 6.

2.3. Языки высокого уровня: алфавит, синтаксис, семантика

Алфавит и лексемы

Все тексты на языке пишутся с помощью его **алфавита**. Алфавит языка ПАСКАЛЬ включает в себя:

— прописные и строчные латинские буквы, знак подчеркивания _;
— цифры от 0 до 9;
— специальные символы, например +, *, { и @;

— пробельные символы — пробел, табуляцию и переход на новую строку.

Из символов составляются **лексемы**, т. е. минимальные единицы языка, имеющие самостоятельный смысл:

— константы;
— имена (идентификаторы);
— ключевые слова;
— знаки операций;
— разделители (скобки, точка, запятая, пробельные символы).

Лексемы языка программирования аналогичны словам естественного языка. Например, лексемами являются число 128, имя Vasia, ключевое слово goto и знак операции сложения «+».

Компилятор при синтаксическом разборе текста программы определяет границы одних лексем по другим лексемам, например по разделителям или знакам операций. Из лексем строятся выражения и операторы.

Константы

Константа — величина, не изменяющая свое значение в процессе работы программы (табл. 2.3). Две нижние строки таблицы представляют собой примеры соответствующих констант.

Таблица 2.3

Классификация констант языка ПАСКАЛЬ

Целые		Вещественные		Символьные	Строковые
десятичные	16-ричные	с плавающей точкой	с порядком		
2 15	$0101 $FFA4	–0.26 .005 21.	1.2e4 0.1E–5	‘k’ #186 ЛИ	‘абырвалг’ ’I’m fine’

Имена, ключевые слова и знаки операций

Имена в программах служат той же цели, что и имена людей — чтобы обращаться к программным объектам и различать их, т. е. идентифицировать. Поэтому имена также называют *идентификаторами*.

Как уже говорилось, данные, с которыми работает программа, надо описывать. Для этого служат операторы описания, которые связывают данные с именами. Имена дает программист, при этом следует соблюдать следующие правила:

— имя должно начинаться с буквы;
— имя должно содержать только буквы, знак подчеркивания и цифры;
— прописные и строчные буквы не различаются;
— длина имени практически не ограничена.

Имена даются элементам программы, к которым требуется обращаться, — переменным, константам, процедурам, функциям, меткам и так далее.

Ключевые (зарезервированные) слова — это идентификаторы, которые имеют специальное значение для компилятора. Их можно использовать только в том смысле, в котором они определены. Например, для оператора перехода определено ключевое слово goto, а для описания переменных — var. Имена, создаваемые программистом, не должны совпадать с ключевыми словами.

Знак операции — это один или более символов, определяющих действие над операндами. Внутри знака операции пробелы не допускаются. Например, операция сравнения на «меньше или равно» обозначается <=, а целочисленное деление записывается как div. Операции делятся на унарные (с одним операндом) и бинарные (с двумя).

2.4. Концепция типа данных

Что определяет тип данных?

Данные, с которыми работает программа, хранятся в оперативной памяти. Естественно, что необходимо точно знать, сколько места они занимают, как именно закодированы и какие действия с ними можно выполнять. Все это задается при описании данных с помощью типа. Тип данных однозначно определяет: внутреннее представление данных, а следовательно, и диапазон их возможных значений; допустимые действия над данными (операции и функции). Например, целые и вещественные числа, даже если они занимают одинаковый объем памяти, имеют совершенно разный диапазон возможных значений; целые числа можно умножать друг на друга, а, к примеру, символы — нельзя.

Каждое выражение в программе имеет определенный тип (табл. 2.4).

Таблица 2.4

Классификация типов

Стандартные	Определяемые программистом		
	Простые	Составные	
Логические. Целые. Вещественные. Символьный. Строковый. Адресный. Файловые	Перечисляемый. Интервальный. Адресные	Массивы. Строки. Записи. Множества	Файлы. Процедурные типы. Объекты

Стандартные типы не требуют предварительного определения. Для каждого типа существует ключевое слово, которое используется при описании переменных, констант и т. д. Если же программист определяет собственный тип данных, он описывает его характеристики и сам дает ему имя, которое затем применяется точно так же, как имена стандартных типов. Типы, выделенные в таблице подчеркиванием, объединяются термином «порядковые».

Логические типы

Основной логический тип данных языка ПАСКАЛЬ называется boolean. Величины этого типа занимают в памяти 1 байт и могут принимать всего два значения: true (истина) или false (ложь). Внутреннее представление значения false — 0 (нуль), значения true — 1.

К величинам логического типа применяются логические операции and, or, xor и not (табл. 2.5). Для наглядности вместо значения false используется 0, а вместо true — 1.

Таблица 2.5

Логические операции

a	*b*	*a* and *b*	*a* or *b*	*a* xor *b*	not *a*
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Операция and — «логическое И», логическое умножение. Операция or — «логическое ИЛИ», логическое сложение. Операция xor — так называемое исключающее ИЛИ, или операция неравнозначности. Логическое отрицание not является унарной операцией.

Кроме этого, величины логического типа можно сравнивать между собой с помощью **операций отношения** (табл. 2.6). Результат этих операций имеет логический тип.

Таблица 2.6

Операции отношения

Операция	Знак операции
Больше	>
Больше или равно	>=
Меньше	<
Меньше или равно	<=
Равно	=
Не равно	<>

Целые типы

Целые числа представляются в компьютере в двоичной системе счисления. В языке ПАСКАЛЬ, определено несколько целых типов данных, отличающихся длиной и наличием знака (табл. 2.7).

Таблица 2.7

Целые типы данных

Тип	Название	Размер	Знак	Диапазон значений
Integer	Целое	2 байта	Есть	–32 768 ÷ 32 767 $-2^{15} \div (2^{15} - 1)$
Shortint	Короткое Целое	1 байт	Есть	–128 ÷ 127 $-2^{7} \div (2^{7} - 1)$
Byte	Байт	1 байт	Нет	0 ÷ 255 $0 \div (2^{8} - 1)$
Word	Слово	2 байта	Нет	0 ÷ 65 535 $0 \div (2^{16} - 1)$
Longint	Длинное Целое	4 байта	Есть	–2 147 483 648 ÷ 2 147 483 647 $-2^{31} \div (2^{31} - 1)$

С целыми величинами можно выполнять **арифметические операции** (табл. 2.8). Результат их выполнения всегда целый (при делении дробная часть отбрасывается).

Таблица 2.8

Арифметические операции для величин целых типов

Операция	Знак операции
Сложение	+
Вычитание	–
Умножение	*
Деление	div
Остаток от деления	mod

К целым величинам можно также применять **операции отношения**, перечисленные в разделе «Логические типы». Результат этих операций имеет логический тип.

Кроме этого, к целым величинам можно применять **поразрядные операции** and, or, xor и not. При выполнении этих операций каждая величина представляется как совокупность двоичных разрядов. Действие выполняется над каждой парой соответствующих разрядов операндов. Например, результатом операции 3 and 2 будет 2, поскольку двоичное представление числа 3 — 11, числа 2 — 10.

Для работы с целыми величинами предназначены также **операции сдвига** влево shl и вправо shr. Слева от знака операции указы-

вается, с какой величиной будет выполняться операция, а справа — на какое число двоичных разрядов требуется сдвинуть величину. Например, результатом операции 12 shr 2 будет значение 3, поскольку двоичное представление числа 12 — 1100.

К целым величинам можно применять стандартные функции и процедуры (табл. 2.9).

Таблица 2.9

Стандартные функции и процедуры для величин целых типов

Имя	Описание	Результат	Пояснения
		Функции	
Abs	Модуль	Целый	$\|x\|$ записывается abs(x)
Arctan	Арктангенс угла	Вещественный	arctg x записывается arctan(x)
Cos	Косинус угла	Вещественный	cos x записывается cos(x)
Exp	Экспонента	Вещественный	e^x записывается exp(x)
Ln	Натуральный логарифм	Вещественный	$\log_e x$ записывается ln(x)
Odd	Проверка на четность	Логический	odd (3) даст в результате true
Pred	Предыдущее значение	Целый	pred (3) даст в результате 2
Sin	Синус угла	Вещественный	sin x записывается sin(x)
Sqr	Квадрат	Целый	x^2 записывается sqr(x)
Sqrt	Квадратный корень	Вещественный	$\sqrt{x}$ записывается sqrt(x)
Succ	Следующее значение	Целый	succ(3) даст в результате 4
		Процедуры	
Inc	инкремент	—	inc(x) — увеличить x на 1 inc(x, 3) — увеличить x на 3
Dec	декремент	—	dec(x) — уменьшить x на 1 dec(x, 3) — уменьшить x на 3

Вещественные типы

Вещественные типы данных хранятся в памяти компьютера иначе, чем целые. Внутреннее представление вещественного числа состоит из двух частей — мантиссы и порядка, и каждая часть имеет знак. Например, число 0,087 представляется в виде $0{,}87 \cdot 10^{-1}$, и в памяти хранится мантисса 87 и порядок –1 (для наглядности здесь пренебрегли тем, что данные на самом деле представляются в двоичной системе счисления и несколько сложнее).

Существует несколько вещественных типов, различающихся точностью и диапазоном представления данных (табл. 2.10). Точность числа определяется длиной мантиссы, а диапазон — длиной порядка.

Таблица 2.10

Вещественные типы данных

Тип	Название	Размер, байт	Число значащих цифр	Диапазон значений
Real	Вещественный	6	11—12	$2{,}9 \cdot 10^{-39}$ — $1{,}7 \cdot 10^{38}$
Single	Одинарной точности	4	7—8	$1{,}5 \cdot 10^{-45}$ — $3{,}4 \cdot 10^{38}$
Double	Двойной точности	8	15—16	$5{,}0 \cdot 10^{-324}$ — $1{,}7 \cdot 10^{308}$
Extended	Расширенный	10	19—20	$3{,}4 \cdot 10^{-4932}$ — $1{,}1 \cdot 10^{4923}$
Comp	Большое целое	8	19—20	$-9{,}22 \cdot 10^{18}$ — $9{,}22 \cdot 10^{18}$ -2^{63} — $(2^{63} - 1)$

Для первых четырех типов приведены абсолютные величины диапазонов. С вещественными величинами можно выполнять **арифметические операции** (табл. 2.11). Результат их выполнения — вещественный.

Таблица 2.11

Арифметические операции для величин вещественных типов

Операция	Знак операции
Сложение	+
Вычитание	–
Умножение	*
Деление	/

В общем случае при выполнении любой операции операнды должны быть одного и того же типа, но целые и вещественные величины смешивать разрешается.

К вещественным величинам можно также применять **операции отношения**, перечисленные в разделе «Логические типы». Результат этих операций имеет логический тип.

К вещественным величинам можно применять стандартные функции (табл. 2.12).

Таблица 2.12

Стандартные функции для величин вещественных типов

Имя	Описание	Результат	Пояснения
Abs	Модуль	Вещественный	$\|x\|$ записывается abs(x)
Arctan	Арктангенс угла	Вещественный	arctg x записывается arctan(x)

Окончание табл. 2.12

Имя	Описание	Результат	Пояснения
Cos	Косинус угла	Вещественный	cos *x* записывается cos(x)
Exp	Экспонента	Вещественный	e^x записывается exp(x)
Frac	Дробная часть аргумента	Вещественный	frac(3.1) даст в результате 0,1
Int	Целая часть аргумента	Вещественный	int(3.1) даст в результате 3.0
Ln	Натуральный логарифм	Вещественный	$\log_e x$ записывается ln(x)
Pi	Значение числа π	Вещественный	3,1415926536
Round	Округление до целого	Целый	round(3.1) даст в результате 3, round(3.8) даст в результате 4
Sin	Синус угла	Вещественный	sin *x* записывается sin(x)
Sqr	Квадрат	Вещественный	x^2 записывается sqr(x)
Sqrt	Квадратный корень	Вещественный	$\sqrt{x}$ записывается sqrt(x)
Trunc	Целая часть аргумента	Целый	trunc(3.1) даст в результате 3

Символьный тип

Этот тип данных, обозначаемый ключевым словом char, служит для представления любого символа из набора допустимых символов. Под каждый символ отводится 1 байт. К символам можно применять **операции отношения** (<, <=, >, >=, =, <>), при этом сравниваются коды символов. Меньшим окажется символ, код которого меньше.

Стандартных функций для работы с символами немного (табл. 2.13).

Таблица 2.13

Стандартные функции для величин символьного типа

Имя	Описание	Результат	Пояснения
Ord	Порядковый номер символа	Целый	ord('b') даст в результате 98 ord('ю') даст в результате 238
Chr	Преобразование в символ	Символьный	chr(98) даст в результате «b» chr(238) даст в результате «ю»
Pred	Предыдущий символ	Символьный	pred('b') даст в результате «a»

Окончание табл. 2.13

Имя	Описание	Результат	Пояснения
Succ	Последующий символ	Символьный	succ('b') даст в результате «c»
Upcase	Перевод в верхний регистр	Символьный	upcase('b') даст в результате «B»

Порядковые типы

В группу порядковых объединены целые, символьный, логический, перечисляемый и интервальный типы. Сделано это, потому что они обладают следующими общими чертами:

— все возможные значения порядкового типа представляют собой ограниченное упорядоченное множество;

— к любому порядковому типу может быть применена стандартная функция Ord, которая в качестве результата возвращает порядковый номер конкретного значения в данном типе;

— к любому порядковому типу могут быть применены стандартные функции Pred и Succ, которые возвращают предыдущее и последующее значения соответственно;

— к любому порядковому типу могут быть применены стандартные функции Low и High, которые возвращают наименьшее и наибольшее значения величин данного типа.

2.5. Линейные программы

Линейной называется программа, все операторы которой выполняются последовательно, в том порядке, в котором они записаны. Это самый простой вид программ.

Переменные

Переменная — это величина, которая во время работы программы может изменять свое значение. Все переменные, используемые в программе, должны быть описаны в разделе описания переменных, начинающемся со служебного слова var.

Для каждой переменной задаются ее имя и тип, например:

```
var number : integer;
  x, y     : real;
  option   : char;
```

Имя переменной определяет место в памяти, по которому находится значение переменной. Имя дает программист. Оно должно отражать смысл хранимой величины и быть легко распознаваемым.

Тип переменных выбирается исходя из диапазона и требуемой точности представления данных.

При объявлении можно присвоить переменной некоторое начальное значение, т. е. *инициализировать* ее. Под инициализацией понимается задание значения, выполняемое до начала работы программы. Инициализированные переменные описываются после ключевого слова const:

```
const
  number : integer = 100;
  x      : real = 0.02;
  option : char = 'ю';
```

По умолчанию все переменные, описанные в главной программе, обнуляются.

Выражения

Выражение — это правило вычисления значения. В выражении участвуют *операнды*, объединенные знаками операций. Операндами выражения могут быть константы, переменные и вызовы функций. Операции выполняются в определенном порядке в соответствии с *приоритетами*, как и в математике. Для изменения порядка выполнения операций используются *круглые скобки*, уровень их вложенности практически не ограничен.

Результатом выражения всегда является значение определенного типа, который определяется типами операндов. Величины, участвующие в выражении, должны быть *совместимых типов*.

Далее приведены операции языка ПАСКАЛЬ упорядоченные по убыванию приоритетов.

1. Унарная операция not, унарный минус -, взятие адреса @.
2. Операции типа умножения: * / div mod and shl shr.
3. Операции типа сложения: + - or xor.
4. Операции отношения: = <> < > <= >= in.

Функции, используемые в выражении, вычисляются в первую очередь.

Примеры выражений:

t + sin(x)/2 * x — результат имеет вещественный тип;
a <= b + 2 — результат имеет логический тип;
(x > 0) and (y < 0) — результат имеет логический тип.

Структура программы

Программа на языке ПАСКАЛЬ состоит из необязательного заголовка, разделов описаний и раздела операторов:

```
program имя; {заголовок}
  разделы описаний
begin
  раздел операторов
end.     (* программа заканчивается точкой *)
```

Программа может содержать *комментарии*, заключенные в фигурные скобки {} или в скобки вида (* *).

Общая структура программы приведена на рис. 2.1.

В **разделе операторов** записываются исполняемые операторы программы. Ключевые слова begin и end не являются операторами, а служат для их объединения в так называемый *составной оператор*, или *блок*. Блок может записываться в любом месте программы, где допустим обычный оператор.

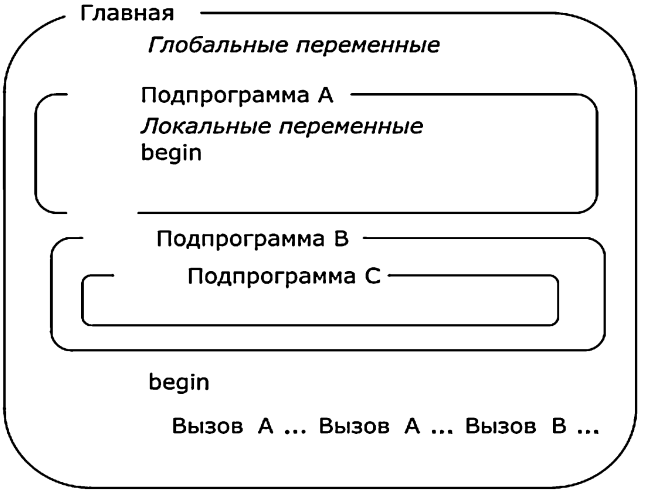

Рис. 2.1. **Общая структура программы на языке ПАСКАЛЬ**

Разделы описаний бывают нескольких видов: описание модулей, констант, типов, переменных, меток, процедур и функций.

Модуль — это подключаемая к программе библиотека ресурсов (подпрограмм, констант и т. п.).

Раздел описания модулей, если он присутствует, должен быть первым. Описание начинается с ключевого слова uses, за которым через запятую перечисляются все подключаемые к программе модули, как стандартные, так и собственного изготовления, например:

```
uses crt, graph, my_module;
```

Количество и порядок следования остальных разделов произвольны, ограничение только одно: любая величина должна быть описана до ее использования. Признаком конца раздела описания является начало следующего раздела. В программе может быть несколько однотипных разделов описаний, но для упрощения структуры программы рекомендуется группировать все однотипные описания в один раздел.

В **разделе описания переменных** необходимо определить все переменные, которые будут использоваться в основной программе.

Раздел описания констант служит для того, чтобы вместо значений констант можно было использовать в программе их имена. Есть и еще одно применение раздела описания констант: в нем описываются переменные, которым требуется присвоить значение до начала работы программы:

```
const weight: real = 61.5;
  n = 10;
```

Раздел описания меток начинается с ключевого слова label, за которым через запятую следует перечисление всех меток, встречающихся в программе. Метка — это либо имя, либо положительное число, не превышающее 9999. Метка ставится перед любым исполняемым оператором и отделяется от него двоеточием. Пример описания меток:

```
label 1, 2, error;
```

Метки служат для организации перехода на конкретный оператор с помощью *оператора безусловного перехода* goto.

Процедуры ввода-вывода

Любая программа при вводе исходных данных и выводе результатов взаимодействует с внешними устройствами. Совокупность стандартных устройств ввода и вывода, т. е. клавиатуры и экрана дисплея, называется консолью.

Ввод с клавиатуры. Для ввода с клавиатуры определены следующие процедуры:

```
read и readln:
read(список);
readln[(список)];
```

В скобках указывается *список имен переменных через запятую*. Квадратные скобки указывают на то, что список может отсутствовать. Например:

```
read(a, b, c);
readln(y);
readln;
```

ВНИМАНИЕ

Вводить можно целые, вещественные, символьные и строковые величины. Вводимые значения должны разделяться любым количеством пробельных символов (пробел, табуляция, перевод строки).

Ввод значения каждой переменной выполняется так:

— значение переменной выделяется как группа символов, расположенных между разделителями;

— эти символы преобразуются во внутреннюю форму представления, соответствующую типу переменной;

— значение записывается в ячейку памяти, определяемую именем переменной.

Кроме этого, процедура readln после ввода всех значений выполняет переход на следующую строку исходных данных. Процедура readln без параметров просто ожидает нажатия клавиши Enter.

Особенность ввода символов и строк состоит в том, что пробельные символы в них ничем не отличаются от всех остальных, поэтому разделителями являться не могут.

Вывод на экран. При выводе выполняется обратное преобразование: из внутреннего представления в символы, выводимые на экран. Для этого в языке определены стандартные процедуры write и writeln:

```
write(список);
writeln[(список)];
```

Процедура write выводит указанные в списке величины на экран, а процедура writeln вдобавок к этому еще и переводит курсор на следующую строку. Процедура writeln без параметров просто переводит курсор на следующую строку.

Выводить можно величины логических, целых, вещественных, символьного и строкового типов. В списке могут присутствовать не только имена переменных, но и выражения, а также их частный случай — константы. Кроме того, для каждого выводимого значения можно задавать его *формат*, например:

```
writeln (a:4, b:6:2);
```

После имени переменной *a* через двоеточие указано количество отводимых под нее позиций, внутри которых значение выравнивается по правому краю. Для *b* указано две форматные спецификации, означающие, что под эту переменную отводится всего шесть позиций, причем две из них — под дробную часть.

Правила записи процедур вывода

Список вывода разделяется запятыми.

Список содержит выражения логических, целых, вещественных, символьного и строкового типов.

После любого значения можно через двоеточие указать количество отводимых под него позиций. Если значение короче, оно прижимается к правому краю отведенного поля, если длиннее, поле «раздвигается» до необходимых размеров.

Для вещественных чисел можно указать второй формат, указывающий, сколько позиций *из общего количества* позиций отводится под дробную часть числа. Необходимо учитывать, что десятичная точка также занимает одну позицию.

Если форматы не указаны, под целое число, символ и строку отводится минимально необходимое для их представления количество позиций. Под вещественное число всегда отводится 17 позиций, причем 10 из них — под его дробную часть.

Форматы могут быть выражениями целого типа.

Пример

Программа, которая переводит температуру в градусах по Фаренгейту в градусы Цельсия по формуле

$$C = \frac{5}{9}(F - 32),$$

где C — температура по Цельсию; F — температура по Фаренгейту.

```
program temperature;
  var fahr, cels : real;                                    {1}
  begin
    writeln('Введите температуру по Фаренгейту');           {2}
    readln(fahr);                                           {3}
    cels := 5 / 9 * (fahr - 32);                            {4}
    writeln('По Фаренгейту: ', fahr:6:2,
            'В градусах Цельсия: ', cels:6:2);              {5}
end.
```

Тема 3
ИНТЕГРИРОВАННЫЕ СРЕДЫ ПРОГРАММИРОВАНИЯ

3.1. Обзор возможностей интегрированных сред

Совокупность средств, с помощью которых программы пишут, корректируют, преобразуют в машинные коды, отлаживают и запускают, называют *средой разработки*, или *оболочкой*. Среда разработки обычно содержит:

— текстовый редактор, предназначенный для ввода и корректировки текста программы;

— компилятор, с помощью которого программа переводится с языка, на котором она написана, в машинные коды;

— средства отладки и запуска программ;

— общие библиотеки, содержащие многократно используемые элементы программ;

— справочную систему и другие элементы.

В качестве примера интегрированной среды рассмотрим Turbo Pascal 7.0.

3.2. Написание, запуск, отладка и корректировка программы

Установка интегрированной среды Turbo Pascal 7.0 (TP) с дистрибутива очень проста, но вряд ли этот дистрибутив доступен кому-то из читателей «за давностью лет». К счастью, в то золотое время для установки у себя на машине программного продукта достаточно было найти у друзей установленную версию, скопировать на дискеты (заархивировав для удобства), переписать к себе на компьютер и разархивировать.

Запуск IDE

Для запуска IDE Turbo Pascal 7.0 надо открыть каталог (папку), в котором расположен файл turbo.exe (обычно это ..\TP\Bin), затем запустить этот файл на исполнение либо двойным щелчком мыши, либо нажатием клавиши Enter.

После запуска появляется рабочий экран, содержащий строку меню, окно редактора и строку состояния.

Строка меню предоставляет доступ к командам интегрированной среды. Для ее активизации нужно нажать клавишу F10, после чего один из элементов меню становится подсвеченным. Перемещение подсветки для выбора нужного элемента меню осуществляется с помощью клавиш со стрелками. После выбора и нажатия клавиши Enter появляется либо выпадающее меню со списком команд, либо окно диалога, соответствующее выбранному элементу.

Окно редактора предназначено для ввода и редактирования текста в одном из исходных файлов программы. Система позволяет одновременно держать в памяти несколько открытых окон, при этом активным является только одно окно, на которое установлен так называемый фокус ввода. Каждое окно имеет *рамку*, в верхней части которой расположены *заголовок окна* (имя файла, возможно с указанием пути к нему) и *номер окна* (окна нумеруются с 1 по 9, в эту нумерацию включается также окно сообщений (Message).

Строка состояния, расположенная в нижней части экрана, содержит контекстно зависимую справочную информацию.

Переключение фокуса ввода с одного окна на другое осуществляется нажатием клавиши F6.

Далее приведено краткое описание назначения каждого элемента меню:

— File — создание, чтение и запись файлов, сохранение внесенных в программу изменений, выход из системы;

— Edit — редактирование текста в активном окне;

— Search — поиск фрагментов текста, контекстная замена и другие операции;

— Run — запуск программы на выполнение в отладочном или обычном режиме;

— Compile — компиляция программы[1], компоновка, задание размещения исполняемого файла;

— Debug — управление возможностями отладчика;

— Tools — работа с окном сообщений и вызов внешних программ, входящих в среду TP;

— Options — установка параметров компиляции, компоновки и других настроек интегрированной среды;

— Window — управление окнами;

— Help — обращение к системе помощи.

ВНИМАНИЕ

Перед первым запуском программы необходимо настроить оболочку. Для этого в пункте меню **Options → Directories** надо ввести полный путь к стандартным модулям языка ПАСКАЛЬ в окно *Unit directories* (найти

[1] То же — для команды Compile из меню Compile.

каталог UNITS внутри каталога, в который установлен TP). Введенные настройки следует сохранить с помощью пункта меню **Options → Save**.

Далее приведены команды меню, достаточные для начала работы с интегрированной средой. Более полную информацию можно получить через меню Help.

Меню File

После выбора элемента меню **File** и нажатия клавиши **Enter** на экране появляется выпадающее меню, содержащее группу команд. Рассмотрим основные команды из этой группы.

Команда **File → New** открывает новое окно редактирования со стандартным именем NONAME00.PAS (вместо 00 используется целое число в диапазоне от 00 до 99). Это имя считается временным (на время ввода нового текста). Если попытаться сохранить набранный текст с помощью команды **File → Save**, то будет вызвано диалоговое окно *Save File As*, в котором предлагается ввести имя файла. Если же на это предложение бездумно нажать **Enter** (не изменяя имя файла), то файл с именем вида NONAME00.PAS будет сохранен в каталоге TP\BIN. Указанный каталог вообще-то предназначен для хранения исполняемых файлов и динамически подключаемых библиотек системы TP, поэтому засорение его какими-либо не относящимися к делу файлами крайне нежелательно. Отсюда первая практическая рекомендация начинающему программисту.

СОВЕТ

Заведите специальную папку (каталог) для размещения ваших программ, создаваемых в среде TP, например D:\TP_WORK, а уже внутри этой папки будете создавать отдельные каталоги для каждой новой программы. О создании такого каталога надо позаботиться еще до начала работы над новой программой. В него вы будете помещать файлы с исходными текстами программы.

После завершения ввода текста в новый файл следует вызвать команду **File → SaveAs**.

Команда **File → SaveAs** вызывает окно диалога *Save File As*. В этом окне выделим следующие элементы:

— текстовое поле *Save File As*, предназначенное для ввода имени файла;

— поле списка *Files*, содержащее список файлов для текущего каталога; в этом поле возможна навигация по списку файлов и каталогов с помощью клавиш со стрелками и клавиши **Enter** — аналогично тому, как это делается в оболочках типа Far или Norton Commander, причем строка с символами ..\ обозначает переход в каталог верхнего уровня (родительский каталог);

— строка состояния (внизу диалогового окна), в которой отображается полный путь к текущему каталогу и текущее имя файла;

— три кнопки: **OK**, **Cancel** и **Help**.

Переход между элементами диалогового окна осуществляется нажатием клавиши **Tab**.

После того как установлен нужный каталог путем навигации в поле *Files*, следует перейти, используя клавишу **Tab**, в текстовое поле *Save File As* и ввести требуемое имя файла (не забывайте контролировать это по строке состояния). Осталось перейти на кнопку **OK** и нажать клавишу **Enter**.

Команда **File → Save (F2)** в зависимости от состояния файла выполняется одним из двух способов:

1) сохраняет на диске текущее состояние файла (после тех изменений, которые в нем сделаны) — если ранее проводилась работа с этим файлом;[1]

2) вызывает окно диалога *Save File As* — если файл абсолютно новый, т. е. создан командой **File → New**.

Команда **File → Open** вызывает окно диалога *Open a File*, которое по составу элементов имеет много общего с окном *Save File As*.

Основная работа по поиску нужного файла проводится в поле *Files*, затем нужно перейти на кнопку **Open** и нажать клавишу **Enter**. В результате текст файла появляется в окне редактирования.

После завершения ввода/редактирования текста в файле следует вызвать команду **File → Save**.

Команда **File → Exit** вызывает завершение работы с оболочкой. Выйти из системы можно и другим способом — нажав комбинацию клавиш **Alt + X**.

Меню Edit

Меню **Edit** позволяет выполнять вырезание, копирование и вставку выделенных фрагментов текста, с которым проводится работа в окне редактирования.

Выделение фрагмента осуществляется разными способами. Самый простой — с применением клавиши **Shift** в комбинации с клавишами со стрелками. Если курсор находится в произвольной позиции строки, то, удерживая нажатой клавишу **Shift**, можно с помощью клавиши → выделить любую подстроку. Если курсор находится в начале строки, то, удерживая нажатой клавишу **Shift**, можно с помощью клавиши ↓ выделить всю строку.

Другой способ выделения фрагментов текста, а также операции по вырезанию, копированию и вставке этих фрагментов реализуются последовательностями команд, задаваемыми сочетаниями клавиш (табл. 3.1).

---

[1] В скобках после обозначения команды меню указывается так называемая «горячая» клавиша (или сочетание клавиш), нажатие которой эквивалентно данной команде.

Таблица 3.1

«Горячие» клавиши для работы с блоком текста

Действие	Последовательность команд
Отметить начало блока	Ctrl + Q B
Отметить конец блока	Ctrl + Q K
Скопировать блок в буфер	Ctrl + Insert
Вставить блок из буфера	Shift + Insert
Вырезать блок, поместив его в буфер	Shift + Delete
Скрыть/отобразить блок	Ctrl + K H

Примечание. Запись Ctrl + Q B означает, что требуется, удерживая клавишу Ctrl, нажать сначала клавишу Q, а затем, отпустив ее, клавишу B.

Команды меню **Edit** реализуют те же самые действия, которые приведены в таблице.

Меню Run

Меню **Run** содержит команды, предназначенные для выполнения программы как в обычном, так и в отладочном режиме.

Команда **Run → Run (Ctrl + F9)** вызывает выполнение откомпилированной ранее программы. Если с момента последней компиляции исходный код был модифицирован, то команда выполнит компиляцию, а затем запустит программу на выполнение.

Остальные команды меню используются при отладке программы.

Команда **Run → Trace into (F7)** осуществляет пошаговое (оператор за оператором) выполнение программы, если при этом встречается вызов функции, то трассировка продолжается с заходом в тело функции и пошаговым выполнением операторов внутри функции.

Команда **Run → Step over (F8)** осуществляет пошаговое выполнение программы, при этом если встречается вызов функции, то функция выполняется как один оператор (без захода в тело функции).

Команда **Run → Go to cursor (F4)** вызывает выполнение программы до того оператора, перед которым установлен текстовый курсор.

Команда **Run → Program reset (Ctrl + F2)** останавливает текущий сеанс отладки, освобождает память, выделенную для программы, и закрывает все открытые файлы.

Команда **Run → Parameters...** позволяет задать текстовую строку параметров, которые DOS передает вызываемой программе.

Эта строка передается программе, находящейся в окне редактора, при ее прогоне.

Меню Compile

Меню **Compile** содержит команды, предназначенные для компиляции программы, находящейся в активном окне.

Команда **Compile → Compile (Alt + F9)** вызывает компиляцию исходного файла (с расширением .pas) в активном окне редактора. Если компиляция прошла успешно, то создается одноименный файл с расширением .exe, если это программа, и .tpu, если это модуль. По умолчанию файл создается в оперативной памяти. Чтобы поместить его на диск, нужно выполнить команду Compile → Destination.

Команда **Compile → Make (F9)** создает исполняемую программу (файл с расширением .exe). При этом перекомпилируются все модули, исходный текст которых был изменен после их компиляции.

Команда **Compile → Build** осуществляет полную перекомпиляцию всех файлов проекта независимо от того, вносились ли в них изменения с момента последней компиляции.

Команда **Compiled Destination** определяет, будет ли результат компиляции помещен на диск или в оперативную память. Переключение между этими режимами выполняется щелчком мыши или нажатием клавиши **Enter**.

Меню Debug

Меню **Debug** содержит команды, управляющие работой встроенного отладчика.

Команда **Debug → Breakpoints...** открывает диалоговое окно *Breakpoints*, позволяющее назначать или отменять точки прерывания, на которых будет останавливаться программа в отладочном режиме. Это окно содержит поле списка *Breakpoint List*, в котором отображается список установленных точек прерывания, и ряд управляющих кнопок, из которых чаще всего используются **OK**, **Edit** и **Delete**. Чтобы добавить новую точку прерывания, надо нажать кнопку **Edit**. Появится диалоговое окно *Breakpoint Modify/New*, содержащее четыре текстовых поля и четыре кнопки: **Modify**, **New**, **Cancel** и **Help**. Текстовые поля предназначены для ввода следующей информации:

— *Condition* — условие, при котором произойдет останов (любое допустимое выражение);

— *Pass Count* — сколько раз точка прерывания пропускается, прежде чем произойдет останов;

— *File Name* — полное имя файла с исходным текстом программы, в котором содержится точка останова;

— *Line Number* — номер строки в этом файле, на которой произойдет останов.

Введя все это и нажав на кнопку **New**, можно получить новую точку прерывания, а строка с указанным номером подсветится красным фоном.

Команда **Debug → Call stack (Ctrl + F3)** открывает диалоговое окно, отображающее стек вызовов — последовательность функций, которые вызывались с момента старта программы.

Команда **Debugs Watch** открывает окно, позволяющее наблюдать значения переменных и выражений, установленных в пункте меню Debug → Add watch...

Команда **Debugs Output** вызывает окно вывода программы.

Команда **Debug → Evaluate/modify (Ctrl + F4) открывает** диалоговое окно с тремя полями: *Expression*, *Result*, *New value*, с помощью которых можно отображать значения переменных (выражений), а также модифицировать значения переменных.

Команда **Debug → Add watch... (Ctrl + F7)** позволяет добавлять новые выражения для просмотра. Если перед вызовом **этого пункта выделить имя переменной в окне редактора, то** оно отобразится в окне, и для его добавления достаточно будет нажать **OK**.

Команда **Debug → Add breakpoint** предназначена для добавления новой точки прерывания программы. При ее выборе появляется то же самое окно, что при нажатии кнопки **Edit** в пункте **Debug → Breakpoints...**

Далее приведена самая простая последовательность отладки при помощи средств оболочки.

1. При помощи пункта меню **Debug → Add** watch установить значения переменных, которые необходимо наблюдать.
2. Открыть окно наблюдаемых значений (**Debug → Watch**), а окно редактора уменьшить так, чтобы окна не перекрывали друг друга.
3. Установить курсор на точку программы, до которой она выполняется правильно.
4. Запустить программу до этой точки (**Run → Go to cursor (F4)**), а затем выполнять ее по шагам (**Run → Trace into (F7)** или **Run → Step over (F8)**), наблюдая в окне значения переменных.
5. Для прерывания процесса отладки используется пункт меню **Run → Program reset (Ctrl + F2)**.

Меню Options

Меню **Options** содержит команды, позволяющие просматривать и модифицировать различные установки (опции) интегрированной среды. Для большинства из этих настроек можно оставить значения, заданные по умолчанию.

В диалоговом окне, появляющемся при выборе команды **Options → Compiler...**, в некоторых случаях (когда TP вдруг забудет, как работать с вещественными числами) может потребоваться отметить пункт 8087/80287.

Команда **Options → Memory Sizes...** может помочь при переполнении стека: с помощью нее задается размер стека, отличный от принятого по умолчанию.

Команда **Options → Directories** требует особого внимания. Четыре поля ввода в этом диалоговом окне позволяют определить четыре группы функциональных каталогов TP:

1) *EXE & TPU Directories* — указывает каталог, в который будут помещаться готовые к работе программы в виде exe-файлов и результат компиляции модулей в виде tpu-файлов;

2) *Include Directories* — содержит те каталоги, в которых TP будет искать включаемые файлы, т. е. файлы, задаваемые директивой компилятора;

3) *Unit Directories* — задает каталоги, в которых среда ищет tpu-файлы, если они не обнаружены в текущем каталоге;

4) если в программе используются внешние процедуры и функции, они должны быть представлены в виде obj-файлов. Поле *Object Directories* задает один или несколько каталогов, в которых TP будет искать эти файлы, если их нет в текущем каталоге.

После того как закончена работа с настройками среды, их следует сохранить, выполнив команду **Options → Save.**

Меню Window

Меню **Window** содержит команды управления окнами. Назначение большинства из них понятно из их названий. Например, пункт **Zoom (F5)** открывает окно на весь экран (это удобно при пользовании справкой), пункт **Next** служит для перехода к следующему открытому окну (для этого удобнее пользоваться клавишей **F6**), а пункт List... открывает доступ ко всем окнам, открытым в оболочке в данный момент. Назначение каждого пункта можно уточнить, выделив команду и нажав клавишу **F1**. Будет вызвана встроенная справочная помощь.

Меню Help

Меню **Help** содержит команды, через которые можно получить доступ к различным частям справочной системы TP. Несмотря на то что справка написана по-английски, даже те, кто не изучал этот язык, смогут найти для себя полезные сведения. Удобнее всего пользоваться контекстной справкой: нажать **CTRL + F1**, когда курсор установлен в окне редактора на имени интересующей стандартной функции, ключевом слове или другом элементе.

Тема 4
СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ

4.1. Базовые конструкции структурного программирования и их реализация в виде управляющих конструкций языка

Главное требование, которому должна удовлетворять программа, — работать в полном соответствии со спецификацией и адекватно реагировать на любые действия пользователя. Кроме этого, программа должна быть выпущена точно к заявленному сроку и допускать оперативное внесение необходимых изменений и дополнений. Объем занимаемой памяти и эффективность алгоритмов при этом, к сожалению, отходят на второй план.

Иными словами, современные критерии качества программы — это прежде всего надежность, а также возможность точно планировать производство программы и ее сопровождение. Для достижения этих целей программа должна иметь простую структуру, быть хорошо читаемой и легко модифицируемой.

Структурное программирование — это технология создания программ, позволяющая путем соблюдения определенных правил уменьшить время разработки и количество ошибок, а также облегчить возможность модификации программы. Структурный подход охватывает все стадии разработки проекта: спецификацию, проектирование, собственно программирование и тестирование.

В теории программирования доказано, что программу для решения задачи любой сложности можно составить только из трех структур, называемых следованием, циклом и ветвлением. *Следованием* называется конструкция, представляющая собой последовательное выполнение двух или более операторов (простых или составных) (рис. 4.1, *а*). *Цикл* задает многократное выполнение оператора (рис. 4.1, *б*). *Ветвление* задает выполнение либо одного, либо другого оператора в зависимости от выполнения какого-либо условия (рис. 4.1, *в*).

Следование, ветвление и цикл называют *базовыми конструкциями структурного программирования*. Их особенностью является то, что любая из них имеет только один вход и один выход, поэтому они могут вкладываться друг в друга. Например, цикл может содер-

жать следование из двух ветвлений, каждое из которых включает вложенные циклы.

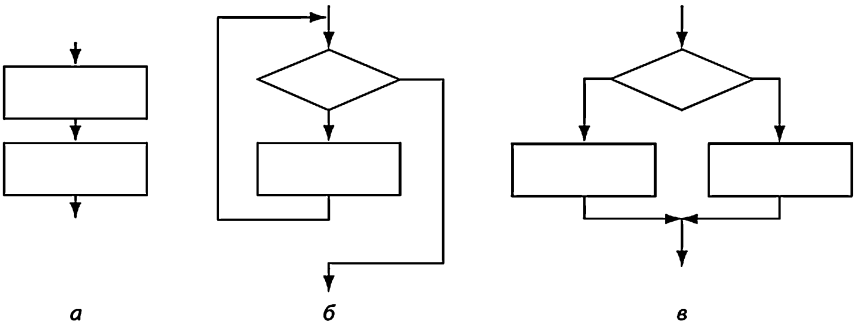

Рис. 4.1. **Базовые конструкции структурного программирования:**
а — следование; *б* — цикл; *в* — ветвление

Целью использования базовых конструкций является получение программы простой структуры. Такую программу легко читать, отлаживать и при необходимости вносить в нее изменения. Язык ПАСКАЛЬ способствует созданию хорошо структурированных программ, поскольку базовые конструкции реализуются в нем непосредственно с помощью соответствующих операторов.

4.2. Программирование условий: условный оператор, оператор выбора

Условный оператор if

Условный оператор if используется для разветвления процесса вычислений на два направления. Структурная схема оператора приведена на рис. 4.2.

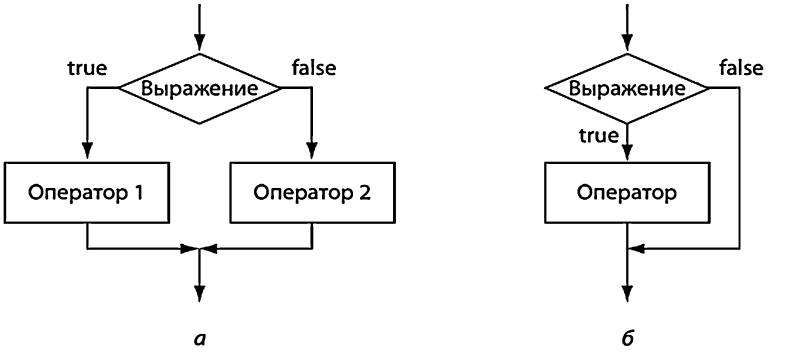

Рис. 4.2. **Структурная схема условного оператора:**
а — условный оператор с двумя ветвями;
б — сокращенный условный оператор

Формат оператора:

```
if выражение then оператор_1
[else оператор_2;]
```

Сначала вычисляется выражение, которое должно иметь логический тип. Если оно имеет значение true, выполняется первый оператор, иначе — второй. После этого управление передается на оператор, следующий за условным.

Одна из ветвей может отсутствовать. Операторы, входящие в состав условного, могут быть простыми или составными. *Составной оператор* (*блок*) обрамляется ключевыми словами begin и end. Блок применяют в том случае, когда по какой-либо ветви требуется выполнить несколько операторов.

Примеры:

```
if a < 0 then b:= 1;                                    {1}

if (a < b) and ((a > d) or (a = 0)) then inc (b)
else begin
  b:= b * a; a:= 0
end;                                                    {2}
if a < b then
  if a < c then m:= a else m:= c
else
  if b < c then m:= b else m:= c;                       {3}
```

Если требуется проверить несколько условий, их объединяют знаками логических операций. Так, выражение в примере {2} будет истинно в том случае, если выполнится одновременно условие $a < b$ и хотя бы одно из условий $a > d$ или $a = 0$.

Частая ошибка при программировании условных операторов — *неверная запись проверки на принадлежность диапазону*. Например, условие $0 < x < 1$ нельзя записать непосредственно. Правильный способ: if (0 < x) and (x < 1) then..., поскольку фактически требуется задать проверку выполнения одновременно двух условий: $x > 0$ и $x < 1$.

Вторая ошибка — отсутствие блока после else, если на самом деле по этой ветви требуется выполнить более одного действия. Эта ошибка не может быть обнаружена компилятором, поскольку является не синтаксической, а семантической, т. е. смысловой.

Пример

Написать программу, которая по введенному значению аргумента вычисляет значение функции, заданной в виде графика (рис. 4.3). Сначала составим описание алгоритма в неформальном словесном виде.

1. Ввести значение аргумента x.

2. Определить, какому интервалу из области определения функции оно принадлежит и вычислить значение функции *y* по соответствующей формуле.

3. Вывести значения *x* и y.

Второй пункт алгоритма следует детализировать. Сначала запишем определение функции в виде формул:

$$y=\begin{cases}0, & x<-2\\ -x-2, & -2\le x<-1\\ x, & -1\le x<1\\ -x+2, & 1\le x<2\\ 0, & x\ge 2\end{cases}.$$

Рис. 4.3. **Функция, заданная в виде графика**

Теперь в соответствии с формулами опишем последовательность действий словами:

— если $x<-2$, то присвоить переменной *y* значение 0;

— если $-2<x<-1$, то присвоить переменной *y* значение $-x-2$;

— если $-1<x<1$, то присвоить переменной *y* значение *x*; и т. д.

```
program calc_function_1;
var x, y: real;
begin
writeln ('Введите значение аргумента'); readln (x);
if x < –2                  then y:= 0;
if (x >= –2) and (x < –1)  then y:= -x - 2;
if (x >= –1) and (x < 1)   then y:= x;
if (x >= 1)  and (x < 2)   then y:= -x + 2;
if x >= 2                  then y:= 0;
writeln ('Для x = ', x:6:2, ' значение функции y = ', y:6:2);
end.
```

Тестовые примеры для этой программы должны включать в себя по крайней мере по одному значению аргумента из каждого интервала, а для проверки граничных условий еще и все точки перегиба.

Оператор варианта case

Оператор варианта (выбора) предназначен для разветвления процесса вычислений на несколько направлений. Структурная схема оператора приведена на рис. 4.4.

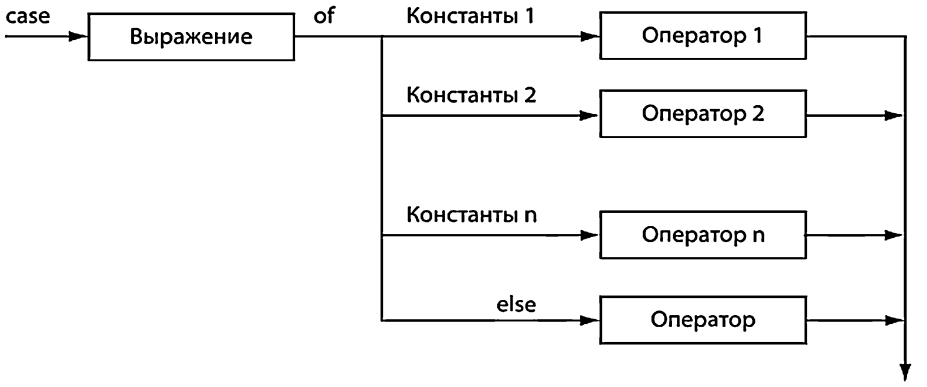

Рис. 4.4. **Структурная схема оператора выбора**

Формат оператора:

```
case выражение of
  константы_1: оператор_1;
  константы_2: оператор_2;
  ...
  константы_n: оператор_n;
  [else: оператор]
end;
```

Выполнение оператора начинается с вычисления выражения. Затем управление передается на оператор, помеченный константами, значение одной из которых совпало с результатом вычисления выражения. После этого выполняется выход из оператора. Если совпадения не произошло, выполняются операторы, расположенные после слова else, а при его отсутствии управление передается оператору, следующему за case.

Выражение после ключевого слова case должно быть порядкового типа, константы — того же типа, что и выражение. Чаще всего после case используется имя переменной. Перед каждой ветвью оператора можно записать одну или несколько констант через запятую или операцию диапазона, обозначаемую двумя идущими подряд точками, например:

```
case a of
4 : writein ('4');
5, 6 : writein ('5 или 6');
7..12 : writein ('от 7 до 12');
end;
```

ВНИМАНИЕ

Если по какой-либо ветви требуется записать не один, а несколько операторов, они заключаются в блок с помощью ключевых слов begin и end.

Пример

Программа, определяющая, какая из курсорных клавиш была нажата.

Для объяснения этой программы надо забежать немного вперед и рассказать о том, что в состав оболочек языка ПАСКАЛЬ входят так называемые модули — библиотеки полезных при программировании ресурсов. В модуле Crt есть функция readkey, позволяющая получить код нажатой клавиши.

Функция readkey работает так: если нажата алфавитно-цифровая клавиша, функция возвращает соответствующий символ. Если нажата клавиша курсора, возвращается символ с кодом 0, а при повторном вызове можно получить так называемый расширенный код клавиши. Для простоты можно считать, что расширенный код — это номер клавиши на клавиатуре. Функция ord позволяет получить числовой код символа.

```
program cursor_keys;
uses Crt;
var key : char;
begin
  writeln('Нажмите одну из курсорных клавиш ');
  key := readkey;
  if ord(key) <> 0 then writeln('обычная клавиша')
  else begin
    key := readkey;
    case ord(key) of
      77: writeln('стрелка вправо');
      75: writeln('стрелка влево');
      72: writeln('стрелка вверх');
      80: writeln('стрелка вниз');
      else writeln('не стрелка');
    end;
  end;
end.
```

4.3. Программирование циклов

Операторы цикла используются для вычислений, повторяющихся многократно. В языке ПАСКАЛЬ три вида циклов: цикл с предусловием while, цикл с постусловием repeat и цикл с параметром for. Каждый из них состоит из определенной последовательности операторов.

Блок, ради выполнения которого и организуется цикл, называется *телом цикла*. Остальные операторы служат для управления процессом повторения вычислений: это начальные установки, проверка условия продолжения цикла и модификация параметра цикла (рис. 4.5). Один проход цикла называется *итерацией*.

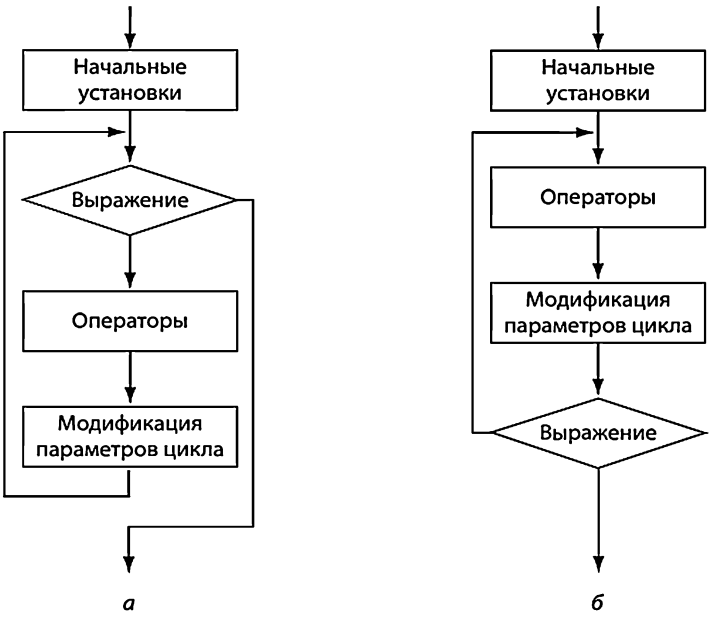

Рис. 4.5. **Структурные схемы операторов цикла:**
а — цикл с предусловием; *б* — цикл с постусловием

Начальные установки служат для того, чтобы до входа в цикл задать значения переменных, которые в нем используются.

Проверка условия продолжения цикла выполняется на каждой итерации либо до тела цикла, тогда говорят о *цикле с предусловием* (см. рис. 4.5, *а*), либо после тела цикла — *цикл с постусловием* (см. рис. 4.5, *б*). Разница между ними состоит в том, что тело цикла с постусловием всегда выполняется хотя бы один раз, после чего проверяется, надо ли его выполнять еще раз. Проверка необходимости выполнения цикла с предусловием делается до тела цикла, поэтому возможно, что он не выполнится ни разу.

Параметром цикла называется переменная, которая используется при проверке условия цикла и принудительно изменяется на каждой итерации, причем, как правило, на одну и ту же величину. Если параметр цикла целочисленный, он называется *счетчиком цикла*. Число повторений такого цикла можно определить заранее. Параметр есть не у всякого цикла.

Цикл завершается, если условие его продолжения не выполняется. Возможно принудительное завершение как текущей итерации, так и цикла в целом. Для этого служат операторы break, continue и goto.

Цикл с предусловием while

Формат оператора прост:

```
while выражение do оператор
```

Выражение должно быть логического типа. Например, это может быть операция отношения или просто логическая переменная. Если результат вычисления выражения равен true, выполняется расположенный после служебного слова do простой или составной оператор. Эти действия повторяются до того момента, пока результатом выражения не станет значение false. После окончания цикла управление передается на следующий за ним оператор.

ВНИМАНИЕ

Если в теле цикла необходимо выполнить более одного оператора, необходимо заключить их в блок с помощью begin и end.

Пример

Написать программу печати таблицы значений функции

$$y=\begin{cases} t, & x<0 \\ tx, & 0\le x<10 \\ 2t, & x\ge 10 \end{cases}$$

для аргумента, изменяющегося в заданных пределах с заданным шагом.

Опишем алгоритм в словесной форме.

1. Взять первое значение аргумента.
3. Определить, какому из интервалов оно принадлежит.
4. Вычислить значение функции по соответствующей формуле.
5. Вывести строку таблицы.
6. Перейти к следующему значению аргумента.
7. Если оно не превышает конечное значение, повторить шаги 3—6, иначе закончить.

Шаги 3—6 повторяются многократно, поэтому для их выполнения надо организовать цикл. Назовем в программе начальное значение аргумента X_n, конечное значение аргумента $X_к$, шаг изменения аргумента dX, параметр — t. Все величины вещественные. Программа выводит таблицу, состоящую из двух столбцов — значений аргумента и соответствующих им значений функции.

ПРИМЕЧАНИЕ

Из эстетических соображений можно заменить символы, которыми выполняется графление таблицы, на псевдографические. Чтобы ввести символ с помощью его кода, надо нажать левую клавишу Alt и, не отпуская ее, ввести код символа на дополнительной клавиатуре. При отпускании клавиши Alt символ отобразится на экране.

```
program tabl_fun;
var Xn, Xk, dX, t, x, y : real;
begin
  writeln('Введите Xn, Xk, dX, t');
  readln(Xn, Xk, dX, t);
  writeln(' -------------------------- ');
  writeln('|     X      |      Y      |');
  writeln(, -------------------------- ,);
  x := Xn;                          {Начальные установки}
```

```
  while x <= Xk do begin          {Заголовок цикла}
    if x < 0 then y := t;
    if (x >= 0) and (x < 10) then y := t * x;
    if x >= 10  then y := 2 * t;
    writeln('|', x:9:2,'    |', y:9:2,'    |');
    x := x + dX;                  {Модификация параметра цикла}
  end;
  writeln(' -------------------------- ');
end.
```

Цикл с постусловием repeat

Тело цикла с постусловием заключено между служебными словами repeat и until, поэтому заключать его в блок не требуется:

```
repeat
  тело цикла
until выражение
```

В отличие от цикла while, этот цикл будет выполняться до тех пор, пока логическое выражение после слова until *ложно*. Как только результат выражения станет истинным, произойдет выход из цикла. Вычисление выражения выполняется в конце каждой итерации цикла.

Этот вид цикла применяется в тех случаях, когда тело цикла необходимо обязательно выполнить хотя бы один раз.

Пример

Написать программу, вычисляющую квадратный корень вещественного аргумента X с заданной точностью eps по итерационной формуле

$$Y_n = \frac{1}{2}(Y_{n-1} + X / Y_{n-1}),$$

где Y_{n-1} — предыдущее приближение к корню (в начале вычислений выбирается произвольно); Y_n — последующее приближение. Процесс вычислений прекращается, когда приближения станут отличаться друг от друга по абсолютной величине менее чем на величину заданной точности.

```
program square_root;
var X, eps,          {аргумент и точность}
    Yp, Y : real;    {предыдущее и последующее приближение}
begin
  repeat
    writeln('Введите аргумент и точность (больше нуля): ');
    readln(X, eps);
  until (X > 0) and (eps > 0);
  Y := 1;
  repeat
    Yp := Y;
    Y := (Yp + X / Yp) / 2;
  until abs(Y - Yp) < eps;
  writeln('Корень из ', X:6:3, ' с точностью ', eps:7:5,
'равен', Y:9:5);
end.
```

Цикл с параметром for

Этот оператор применяется, если требуется выполнить тело цикла заранее заданное количество раз. Параметр порядкового типа на каждом проходе цикла автоматически либо увеличивается, либо уменьшается на единицу:

```
for параметр:= выражение_1 to выражение_2 do оператор
for параметр:= выражение_2 downto выражение_1 do оператор
```

Выражения должны быть того же типа, что и переменная цикла, оператор — простым или составным.

Пример

Программа выводит на экран в столбик числа от 10 до 1 и подсчитывает их сумму:

```
var i, sum: integer;
begin
sum:= 0;
for i:= 10 downto 1 do begin
  writeln (i); inc (sum, i)
end;
writeln ('Сумма чисел: ', sum);
end.
```

ВНИМАНИЕ

Если в теле цикла необходимо выполнить более одного оператора, необходимо заключить их в блок с помощью begin и end.

Выражения, определяющие начальное и конечное значения счетчика, вычисляются один раз до входа в цикл. Цикл for реализован в языке ПАСКАЛЬ как цикл с предусловием, т. е. его можно представить в виде эквивалентного оператора while. После нормального завершения цикла значение счетчика не определено.

Рекомендации по использованию циклов

Часто встречающимися *ошибками при программировании циклов* являются использование в теле цикла переменных, которым не были присвоены начальные значения, а также неверная запись условия продолжения цикла. Нужно помнить и о том, что в операторе while истинным должно являться условие повторения вычислений, а в операторе repeat — условие их окончания.

Чтобы избежать ошибок, рекомендуется:

— не забывать о том, что, если в теле циклов while и for требуется выполнить более одного оператора, нужно заключать их в блок;

— убедиться, что всем переменным, встречающимся в правой части операторов присваивания в теле цикла, до этого присвоены значения, а также возможно ли выполнение других операторов;

— проверить, изменяется ли в теле цикла хотя бы одна переменная, входящая в условие продолжения цикла;

— предусматривать аварийный выход из итеративного цикла по достижении некоторого предельно допустимого числа итераций.

Процедуры завершения цикла и оператор передачи управления

В языке ПАСКАЛЬ есть несколько стандартных процедур, изменяющих последовательность выполнения операторов:

— break — завершает выполнение цикла, внутри которого записана;

— continue — выполняет переход к следующей итерации цикла;

— exit — выходит из программы или подпрограммы, внутри которой записана;

— halt — немедленно завершает выполнение программы.

Кроме того, для передачи управления используется оператор перехода goto.

Оператор перехода goto

Этот оператор имеет простой синтаксис: в точке программы, из которой требуется организовать переход, после слова goto через пробел записывается имя метки, например goto 1 или goto error.

При программировании на языке ПАСКАЛЬ необходимость в применении оператора перехода возникает в очень ограниченном количестве ситуаций, в большинстве же случаев используются операторы циклов вместе с процедурами передачи управления.

Использование оператора безусловного перехода оправдано, как правило, в двух случаях:

1) принудительный выход вниз по тексту программы из нескольких вложенных циклов или переключателей;

2) переход из нескольких мест программы в одно (например, если перед выходом из программы необходимо всегда выполнять какие-либо действия).

Во всех остальных случаях следует привести алгоритм к структурному виду, т. е. преобразовать его так, чтобы он мог быть записан с помощью базовых конструкций.

4.4. Средства организации модульности в языках высокого уровня

С увеличением объема программы становится невозможным удерживать в памяти все детали. Естественным способом борьбы со сложностью любой задачи является ее разбиение на части. В языках высокого уровня задача может быть разделена на более простые

и обозримые с помощью подпрограмм, после чего программу можно рассматривать в более укрупненном виде — на уровне их взаимодействия.

Разделение программы на подпрограммы позволяет также избежать избыточности кода, поскольку подпрограмму записывают один раз, а вызывать ее на выполнение можно многократно из разных точек программы.

Следующим шагом в повышении уровня абстракции программы является группировка подпрограмм и связанных с ними данных в отдельные файлы (модули), компилируемые раздельно. *Интерфейсом модуля* являются заголовки всех подпрограмм и описания доступных извне типов переменных и констант. Разбиение на модули уменьшает время перекомпиляции и облегчает процесс отладки, скрывая несущественные детали за интерфейсом модуля и позволяя отлаживать программу по частям (при этом, возможно, разным программистам).

При проектировании программы часто применяется *технология нисходящего проектирования*, основная идея которого состоит в разбиении задачи на подзадачи меньшей сложности, пригодные для рассмотрения по отдельности. Эта технология кратко описана в теме 6.

Подпрограммы

Подпрограммы нужны для того, чтобы упростить структуру программы и облегчить ее отладку. В виде подпрограмм оформляются логические законченные части программы.

Подпрограмма — это фрагмент кода, к которому можно обратиться по имени. Она описывается один раз, а вызываться может столько раз, сколько необходимо. Одна и та же подпрограмма может обрабатывать различные данные, переданные ей в качестве аргументов.

В языке ПАСКАЛЬ два вида подпрограмм: процедуры и функции. Они имеют незначительные отличия в синтаксисе и правилах вызова. Процедуры и функции описываются в соответствующих разделах описания, до начала блока исполняемых операторов.

Само по себе описание не приводит к выполнению подпрограммы. Для того чтобы подпрограмма выполнилась, ее надо вызвать. *Вызов* записывается в том месте программы, где требуется получить результаты работы подпрограммы. Подпрограмма вызывается по имени, за которым следует *список аргументов* в круглых скобках. Если аргументов нет, скобки не нужны. Список аргументов при вызове как бы накладывается на список параметров, поэтому они должны попарно соответствовать друг другу.

Процедура вызывается с помощью отдельного оператора, а функция — в правой части оператора присваивания, например:

```
inc (i); writein (a, b, c); {вызовы процедур}
y:= sin (x) + 1; {вызов функции}
```

Внутри подпрограмм можно описывать другие подпрограммы. Они доступны только из той подпрограммы, в которой они описаны. Рассмотрим правила описания подпрограмм.

Процедуры

Структура процедуры аналогична структуре основной программы:

```
procedure имя [(список параметров)];  {заголовок}
  разделы описаний
begin
  раздел операторов
end;
```

Пример

Найти разность средних арифметических значений двух вещественных массивов из 10 элементов.

Как видно из условия, для двух массивов требуется найти одну и ту же величину — среднее арифметическое. Следовательно, логичным будет оформить его нахождение в виде подпрограммы, которая сможет работать с разными массивами.

```
program dif_average;
const n = 10;
type mas = array[1 .. n] of real;
var  a, b : mas;
    i : integer;
    dif, av_a, av_b : real;

procedure average(x : mas; var av : real);  {1}
  var i : integer;
  begin
    av := 0;
    for i := 1 to n do av := av + x[i];
    av := av / n;
  end;                                      {2}

begin
  for i := 1 to n do read(a[i]);
  for i := 1 to n do read(b[i]);
  average(a, av_a);                         {3}
  average(b, av_b);                         {4}
  dif := av_a – av_b;
  writeln('Разность значений ', dif:6:2)
end.
```

Описание процедуры average расположено в строках с {1} по {2}. В строках, помеченных цифрами {3} и {4}, эта процедура вызывается сначала для обработки массива *a*, затем — массива *b*. Эти массивы передаются в качестве аргументов. Результат вычисления среднего арифметического возвращается в главную программу через второй параметр процедуры.

Функции

Описание функции отличается от описания процедуры незначительно:

```
function имя [(список параметров)] : тип; {заголовок}
  разделы описаний
begin
  раздел операторов
  имя := выражение;
end;
```

Функция вычисляет одно значение, которое передается через ее имя. Следовательно, в заголовке должен быть описан тип этого значения, а в теле функции — оператор, присваивающий вычисленное значение ее имени.

Пример

Найти разность средних арифметических значений двух вещественных массивов из 10 элементов.

```
program dif_averagei;
const n = 3;
type mas = array [1.. n] of real;
var
a, b : mas;
i : integer;
dif : real;
function average (x: mas): real;  {1}
var i : integer;                  {2}
av : real;
begin
  av:= 0;
  for i:= 1 to n do av := av + x[i];
  average:= av / n;               {3}
end;
begin
  for i:= 1 to n do read (a [i]);
  for i:= 1 to n do read (b [i]);
  dif := average (a) – average (b); {4}
writeln ('Разность значений ', dif:6:2)
end.
```

Оператор, помеченный комментарием {1}, представляет собой заголовок функции. Тип функции определен как вещественный, потому что такой тип имеет среднее арифметическое элементов вещественного массива. Оператор {3} присваивает вычисленное значение имени функции. В операторе {4} функция вызывается дважды: сначала для одного массива, затем для другого.

Глобальные и локальные переменные

В IBM PC-совместимых компьютерах память условно разделена на так называемые сегменты. Адрес каждого байта составляется

из номера сегмента и смещения относительно его начала. Компилятор языка ПАСКАЛЬ формирует сегмент кода, в котором хранится программа в виде машинных команд, сегмент данных, в котором выделяется память под глобальные переменные программы, и сегмент стека, предназначенный для размещения локальных переменных во время выполнения программы (рис. 4.6).

Рис. 4.6. **Структура исполняемой программы в оперативной памяти**

Глобальными называются переменные, описанные в главной программе. Переменные, которые не были инициализированы явным образом, перед началом выполнения программы обнуляются. Время жизни глобальных переменных — с начала программы и до ее завершения. Глобальные переменные доступны в любом месте программы или подпрограммы, кроме тех подпрограмм, в которых описаны локальные переменные с такими же именами.

Внутри подпрограмм описываются *локальные переменные*. Они располагаются в сегменте стека, причем распределение памяти происходит в момент вызова подпрограммы, а ее освобождение — по завершении подпрограммы. Значения локальных переменных между двумя вызовами одной и той же подпрограммы не сохраняются, и эти переменные предварительно не обнуляются. Локальные переменные могут использоваться только в подпрограмме, в которой они описаны, и всех вложенных в нее.

В подавляющем большинстве случаев для обмена данными между вызывающей и вызываемой подпрограммами предпочтительнее использовать *механизм параметров*. Если все данные передаются подпрограммам через списки параметров, для локализации места ошибки достаточно просмотреть заголовки подпрограмм, а затем — тексты только тех из них, в которые передается интересующая переменная.

ВНИМАНИЕ

Подпрограмму надо писать таким образом, чтобы вся необходимая для ее использования информация содержалась в ее заголовке.

Виды параметров подпрограмм

Список параметров, т. е. величин, передаваемых в подпрограмму и обратно, содержится в ее заголовке. Для каждого параметра обычно задается его имя, тип и способ передачи. Либо тип, либо способ передачи могут не указываться.

Важно запомнить, что в заголовке подпрограммы нельзя вводить описание нового типа, там должны использоваться либо имена стандартных типов, либо имена типов, описанных программистом ранее в разделе type.

Основными видами параметров языке ПАСКАЛЬ являются параметры-значения, параметры-переменные и параметры-константы.

Параметры-значения. Параметр-значение описывается в заголовке подпрограммы следующим образом:

```
имя: тип;
```

Например, передача в процедуру P величины целого типа записывается так:

```
procedure P (x: integer);
```

Имя параметра может быть произвольным. Параметр *x* можно представить себе как локальную переменную, которая получает свое значение из главной программы при вызове подпрограммы. В подпрограмму передается копия значения аргумента.

Механизм передачи следующий: из ячейки памяти, в которой хранится переменная, передаваемая в подпрограмму, берется ее значение и копируется в область сегмента стека, называемую *областью параметров*. Подпрограмма работает с этой копией, следовательно, доступа к ячейке, где хранится сама переменная, не имеет. По завершении работы подпрограммы стек освобождается. Такой способ называется *передачей по значению*.

При вызове подпрограммы на месте параметра, передаваемого по значению, может находиться выражение. Тип выражения должен быть совместим по присваиванию с типом параметра.

Например, если в вызывающей программе описаны переменные

```
var x: integer;
  c: byte;
  y: longint;
```

то следующие вызовы подпрограммы P, заголовок которой описан ранее, будут синтаксически правильными:

```
P (x); P (c); P (y); P (200); P (x div 4 + 1);
```

Недостатками передачи по значению являются затраты времени на копирование параметра, затраты памяти в стеке и опасность его переполнения, когда речь идет о параметрах, занимающих много места, например массивах большого размера. Поэтому более пра-

вильно использовать для передачи в подпрограмму исходных данных параметры-константы.

Параметры-переменные. Признаком параметра-переменной является ключевое слово var перед описанием параметра:

```
var имя: тип;
```

Например, передача в процедуру P параметра-переменной целого типа записывается так:

```
procedure P (var x: integer);
```

При вызове подпрограммы в область параметров копируется не значение переменной, а ее адрес, и подпрограмма через него имеет доступ к ячейке, в которой хранится переменная. Этот способ передачи параметров называется *передачей по адресу*. Подпрограмма работает непосредственно с переменной из вызывающей программы и, следовательно, может ее изменить.

ВНИМАНИЕ

При вызове подпрограммы на месте параметра-переменной может находиться только ссылка на переменную точно того же типа.

Проиллюстрируем передачу параметров-значений и параметров-переменных на примере.

```
var a, b, c, d, e : word;
procedure X(a, b, c : word; var d : word);
  var e : word;
begin
  c := a + b; d := c; e := c;
  writeln ('Подпрограмма:');
  writeln ('c = ', c, ' d = ', d, ' e = ', e);
end;
begin
  a := 3; b := 5;
  x(a, b, c, d);
  writeln ('Главная программа:');
  writeln ('c = ', c, ' d = ', d, ' e = ', e);
end.
```

Результаты работы этой программы приведены ниже.

Подпрограмма:

```
c = 8 d = 8 e = 8.
```

Главная программа:

```
c = 0 d = 8 e = 0.
```

Значение переменной *c* в главной программе не изменилось, поскольку переменная передавалась по значению, а значение переменной *e* не изменилось, потому что в подпрограмме была описана локальная переменная с тем же именем.

Параметры-константы. Параметр-константу можно узнать по ключевому слову const перед описанием параметра:

```
const имя: тип;
```

Это ключевое слово свидетельствует о том, что в пределах подпрограммы данный параметр изменить невозможно. При вызове подпрограммы на месте параметра может быть записано выражение, тип которого совместим по присваиванию с типом параметра. Фактически параметры-константы передаются по адресу, но доступ к ним обеспечивается только для чтения.

Например, передача в процедуру P параметра-константы целого типа записывается так:

```
procedure P (const x: integer);
```

Подведем итоги. Если данные передаются в подпрограмму по значению, их можно изменять, но эти изменения затронут только копию в области параметров и не отразятся на значении переменной в вызывающей программе. Если данные передаются как параметры-константы, то изменять их в подпрограмме нельзя. Следовательно, эти два способа передачи должны использоваться для передачи в подпрограмму исходных данных.

Параметры составных типов (массивы, записи, строки) предпочтительнее передавать как константы, потому что при этом не расходуется время на копирование и место в стеке.

Результаты работы процедуры следует передавать через параметры-переменные, результат функции — через ее имя.

Рекурсивные подпрограммы

Рекурсивной называется подпрограмма, в которой содержится обращение к самой себе. Такая рекурсия называется *прямой*. Есть также *косвенная рекурсия*, когда две или более подпрограмм вызывают друг друга.

При обращении подпрограммы к самой себе происходит то же самое, что и при обращении к любой другой функции или процедуре: в стек записывается адрес возврата, резервируется место под локальные переменные, выполняется передача параметров, после чего управление передается первому исполняемому оператору подпрограммы.

Для завершения вычислений каждая рекурсивная подпрограмма должна содержать хотя бы одну ветвь алгоритма, заканчивающуюся возвратом в вызывающую программу.

Простой пример рекурсивной функции — вычисление факториала. Чтобы получить значение факториала числа n, требуется умножить на n- факториал числа $(n - 1)$. Известно также, что $0! = 1$ и $1! = 1$.

```
function fact (n: byte): longint;
```

```
begin
  if (n = 0) or (n = 1) then fact:= 1
  else fact:= n * fact (n – 1);
end;
```

Рассмотрим, что происходит при вызове этой функции при $n = 3$. В стеке отводится место под параметр n, ему присваивается значение 3, и начинается выполнение функции. Условие в операторе if ложно, поэтому управление передается на ветвь else. Для вычисления выражения n * fact (n - 1) требуется повторно вызвать функцию fact. Для этого в стеке отводится новое место под параметр n, ему присваивается значение 2, и выполнение функции начинается сначала. В третий раз функция вызывается со значением параметра, равным 1, и вот тут-то становится истинным выражение (n = 0) or (n = 1), поэтому происходит возврат из подпрограммы в точку вызова, т. е. на выражение n * fact(n - 1) для $n = 2$. Результат выражения присваивается имени функции и передается в точку ее вызова, т. е. в то же выражение, только теперь происходит обращение к параметру n, равному 3.

Достоинством рекурсии является компактная запись, а *недостатками* — расход времени и памяти на повторные вызовы функции и передачу ей параметров, а главное, опасность переполнения стека.

Модули

Модуль — это подключаемая к программе библиотека ресурсов. Он может содержать описания типов констант, переменных и подпрограмм. В модуль обычно объединяют связанные между собой ресурсы: например, в составе оболочки есть модуль Graph для работы с экраном в графическом режиме.

Модули применяются либо как библиотеки, которые могут использоваться различными программами, либо для разбиения сложной программы на составные части.

Использование модулей позволяет преодолеть ограничение в один сегмент на объем кода исполняемой программы, поскольку код каждого подключаемого к программе модуля содержится в отдельном сегменте.

Модули можно разделить на *стандартные*, которые входят в состав системы программирования, и *пользовательские*, т. е. создаваемые программистом. Чтобы подключить модуль к программе, его требуется предварительно скомпилировать. Результат компиляции каждого модуля хранится на диске в отдельном файле с расширением .tpu.

Описание модулей

Исходный текст каждого модуля хранится в отдельном файле с расширением .pas. Общая структура модуля:

```
unit имя;          {заголовок модуля}
interface          {интерфейсная секция модуля}
  {описание глобальных элементов модуля (видимых извне)}
implementation     {секция реализации модуля}
  {описание локальных (внутренних) элементов модуля}
Begin              {секция инициализации}
{может отсутствовать}
end
```

ВНИМАНИЕ

Имя файла, в котором хранится модуль, должно совпадать с именем, заданным после ключевого слова unit.

Модуль может использовать другие модули, для этого их надо перечислить в операторе uses, который может находиться только непосредственно после ключевых слов interface или implementation.

В *интерфейсной секции* модуля определяют константы, типы данных, переменные, а также заголовки процедур и функций.

В *секции реализации* описываются подпрограммы, заголовки которых приведены в интерфейсной части. Заголовок подпрограммы должен быть или идентичным указанному в секции интерфейса, или состоять только из ключевого слова procedure или function и имени подпрограммы. Для функции также указывается ее тип.

Кроме того, в этой секции можно определять константы, типы данных, переменные и внутренние подпрограммы.

Секция инициализации предназначена для присваивания начальных значений переменным, которые используются в модуле.

Операторы, расположенные в секции инициализации модуля, выполняются перед операторами основной программы.

Для сохранения скомпилированного модуля на диске требуется установить значение пункта Destination меню Compile в значение Disk. Компилятор создаст файл с расширением .tpu.

Пример

Оформим в виде модуля подпрограмму вычисления среднего арифметического значения элементов массива.

```
unit Avrg;
interface
  const n = 10;
  type mas = array[1 .. n] of real;
  procedure average(x : mas; var av : real);
implementation
  procedure average(x : mas; var av : real);
    var i : integer;
  begin
    av := 0;
    for i := 1 to n do av := av + x[i];
```

```
    av := av / n;
  end;
end.
```

Использование модулей

Для использования в программе величин, описанных в интерфейсной части модуля, имя модуля указывается в разделе uses. Можно записать несколько имен модулей через запятую, например:

```
program example;
uses Avrg, Graph, Crt;
```

Поиск модулей выполняется сначала в библиотеке исполняющей системы, затем в текущем каталоге, а после этого — в каталогах, заданных в диалоговом окне *Options/Directories*.

Если в программе описана величина с тем же именем, что и в модуле, то для обращения к величине из модуля требуется перед ее именем указать через точку имя модуля.

Тема 5
СТРУКТУРЫ И ТИПЫ ДАННЫХ

5.1. Абстрактные типы данных: стек, линейный список, двоичное дерево

Любая программа предназначена для обработки данных, от способа организации которых зависят алгоритмы работы, поэтому выбор структур данных должен предшествовать созданию алгоритмов.

Память под данные выделяется либо на этапе компиляции (в этом случае необходимый объем должен быть известен до начала выполнения программы), либо во время выполнения программы.

Если до начала работы с данными невозможно определить, сколько памяти потребуется для их хранения, то память выделяется по мере необходимости отдельными блоками, связанными друг с другом с помощью указателей.

Такой способ организации данных называется *динамическими структурами данных*, поскольку их размер изменяется во время выполнения программы. Из динамических структур в программах чаще всего используются линейные списки, стеки, очереди и бинарные деревья. Они различаются способами связи отдельных элементов и допустимыми операциями. Конкретная реализация динамической структуры может варьироваться, поэтому такие структуры называются *абстрактными*.

Линейные списки

В **линейном списке** каждый элемент связан со следующим и, возможно, с предыдущим. В первом случае список называется *односвязным*, во втором — двусвязным. Если последний элемент связать указателем с первым, получится *кольцевой список*.

Каждый элемент списка содержит *ключ*, идентифицирующий этот элемент. Ключ обычно бывает либо целым числом, либо строкой и является частью поля данных. В качестве ключа в процессе работы со списком могут выступать разные части поля данных. Ключи разных элементов списка могут совпадать.

Над списками можно выполнять следующие операции:

— начальное формирование списка (создание первого элемента);

— добавление элемента в конец списка;
— чтение элемента с заданным ключом;
— вставка элемента в заданное место списка (до или после элемента с заданным ключом);
— удаление элемента с заданным ключом;
— упорядочивание списка по ключу.

Стеки

Стек — простейшая динамическая структура. Добавление элементов в стек и выборка из него выполняются из одного конца, называемого *вершиной стека* (top). Другие операции со стеком не определены. При выборке элемент исключается из стека. На рис. 5.1 изображен первый элемент стека, к которому добавляются еще два элемента.

Говорят, что стек реализует принцип обслуживания LIFO (*Last In — First Out*, последним пришел — первым обслужен). Стеки широко применяются в системном программном обеспечении, компиляторах, в различных рекурсивных алгоритмах.

Очереди

Очередь — это динамическая структура данных, добавление элементов в которую выполняется в один конец, а выборка — из другого конца. Другие операции с очередью не определены. При выборке элемент исключается из очереди. Говорят, что очередь реализует принцип обслуживания FIFO (*First In — First Out*, первым пришел — первым обслужен). В программировании очереди применяются очень широко — например, при моделировании, буферизованном вводе/ выводе или диспетчеризации задач в ОС.

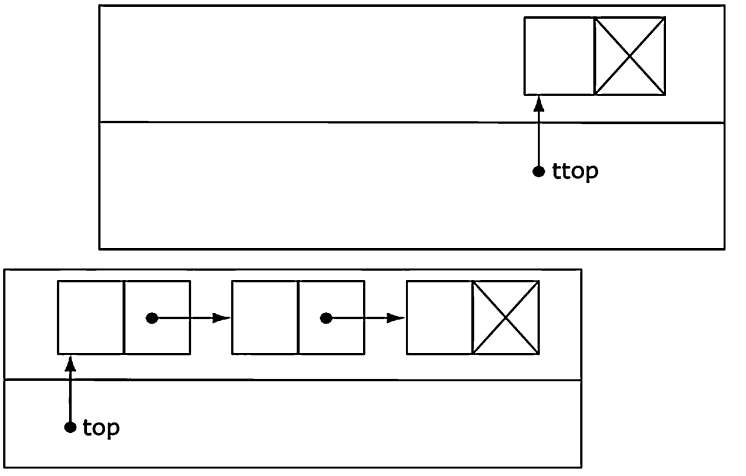

Рис. 5.1. **Добавление двух элементов в стек**

Бинарные деревья

Бинарное дерево — это динамическая структура данных, состоящая из узлов, каждый из которых содержит кроме данных не более двух ссылок на различные бинарные деревья. На каждый узел имеется ровно одна ссылка. Начальный узел называется *корнем дерева*.

Пример бинарного дерева приведен на рис. 5.2 (корень обычно изображается сверху). Узел, не имеющий поддеревьев, называется *листом*. Исходящие узлы называются *предками*, входящие — *потомками*. Высота дерева определяется числом уровней, на которых располагаются его узлы.

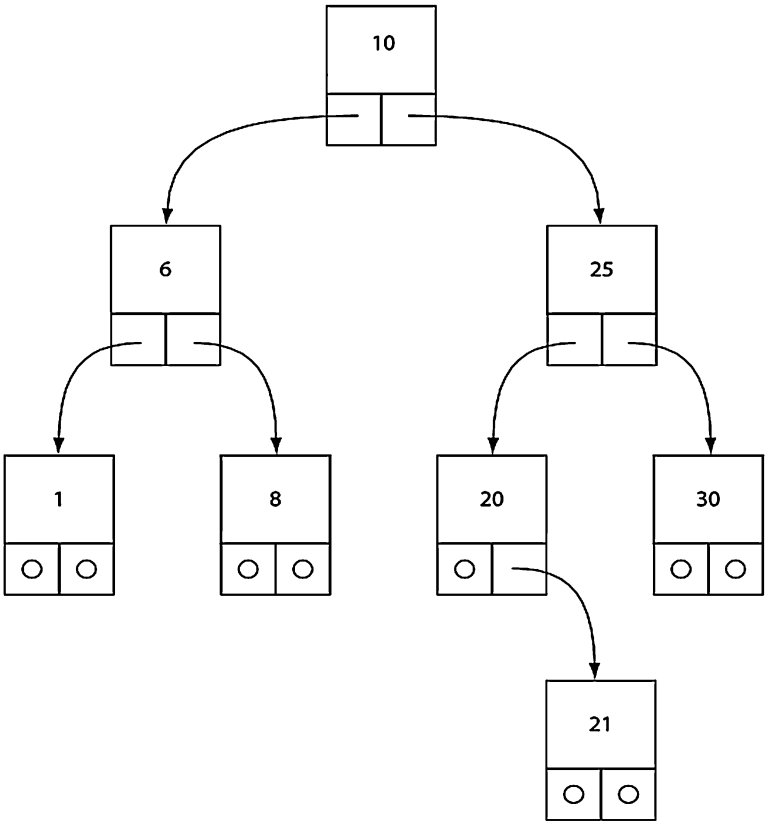

Рис. 5.2. **Пример бинарного дерева**

Если дерево организовано таким образом, что для каждого узла все ключи его левого поддерева меньше ключа этого узла, а все ключи его правого поддерева — больше, оно называется *деревом поиска*. Одинаковые ключи не допускаются. В дереве поиска можно найти элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения ключа в каждом узле. Такой поиск гораздо эффективнее поиска по списку, поскольку

время поиска определяется высотой дерева, а она пропорциональна двоичному логарифму числа узлов.

Для бинарных деревьев определены операции включения узла в дерево, поиска по дереву, обхода дерева, удаления узла.

5.2. Реализация динамических структур средствами языков высокого уровня

В IBM PC-совместимых компьютерах память условно разделена на сегменты. Компилятор формирует сегменты кода, данных и стека, а остальная доступная программе память называется *динамической*, или *хипом*.

Динамические переменные создаются в хипе во время выполнения программы. Обращение к ним осуществляется через *указатели*. С помощью указателей и динамических переменных реализуют динамические структуры данных.

Указатели

Указателем называется переменная, предназначенная для хранения адресов областей памяти. В указателе можно хранить адрес данных или программного кода. Адрес занимает четыре байта и хранится в виде двух слов, одно из которых определяет сегмент, второе — смещение.

Указатели делятся на стандартные и определяемые программистом. Величины *стандартного типа* pointer предназначены для хранения адресов данных произвольного типа:

```
var p : pointer;
```

Программист может определить ***указатель на данные*** или ***подпрограмму конкретного типа***. Как и для других нестандартных типов, это делается в разделе type:

```
type pword = ^o^; {читается как «указатель на word»}
...
var pw: pword;
```

Такие указатели называются *типизированными*. Можно описать указатель на любой тип данных, кроме файловых. Тип указателя на данные можно описать и непосредственно при описании переменной:

```
var pw: ^word;
```

Для указателей определены только операции *проверки на равенство и неравенство и присваивания*.

Указатели присваиваются согласно следующим правилам.

1. Любому указателю можно присвоить стандартную константу nil, которая означает, что указатель не ссылается на какую-либо конкретную ячейку памяти.

2. Указатели стандартного типа pointer совместимы с указателями любого типа.

3. Указателю на конкретный тип данных можно присвоить только значение указателя того же или стандартного типа.

Операция @ и функция addr позволяют получить ***адрес переменной***:

```
var x: word; {переменная}
  pw: ^word; {указатель на величины типа word}
...
pw := @w; {или pw := addr (w);}
```

Для обращения к значению переменной, адрес которой хранится в указателе, применяется ***операция разадресации*** (***разыменования***), обозначаемая с помощью символа «^»:

```
pw^: = 2; inc (pw^); writeln (pw^);
```

С величинами, адрес которых хранится в указателе, можно выполнять любые действия, допустимые для значений этого типа.

Стандартными функциями для работы с указателями являются:

— addr(x) : pointer — возвращает адрес *x* (аналогично операции @), где *x* — имя переменной или подпрограммы;

— seg(x) : word — возвращает адрес сегмента для *x*;

— ofs(x) : word — возвращает смещение для *x*;

— cseg : word — возвращает значение регистра сегмента кода CS;

— dseg : word — возвращает значение регистра сегмента данных DS;

— ptr(seg, ofs : word) : pointer — по заданному сегменту и смещению формирует адрес типа pointer.

Динамические переменные

Динамические переменные создаются в хипе во время выполнения программы с помощью подпрограмм new или getmem. Динамические переменные не имеют собственных имен — к ним обращаются через указатели.

Процедура new (var p : тип_указателя) выделяет в динамической памяти участок размера, достаточного для размещения переменной того типа, на который ссылается указательр, и адрес начала этого участка заносит в этот указатель.

Функция new(тип_указателя) : pointer выделяет в динамической памяти участок размера, достаточного для размещения переменной базового типа для заданного типа указателя, и возвращает адрес начала этого участка; new применяется для типизированных указателей.

Процедура getmem(var p : pointer; size : word) выделяет в динамической памяти участок размером в size байт и присваивает адрес его начала указателю p. Если выделить требуемый объем памяти

не удалось, программа аварийно завершается. Указатель может быть любого типа.

Пример

Рассмотрим работу с динамическими переменными.

```
type pword = ^word;
rec = record
  d: word; s: string;
end;
var p1, p2: pword; p3: ^rec;
```

В разделе исполняемых операторов программы запишем операторы:

```
new (p1); p2:= new (pword); new (p3);
```

В результате выполнения процедуры new(pl) в хипе выделяется объем памяти, достаточный для размещения переменной типа word, и адрес начала этого участка памяти записывается в переменную p1. Второй оператор выполняет аналогичные действия, но используется функция new. При вызове процедуры new с параметром p3 в динамической памяти будет выделено количество байтов, достаточное для размещения записи типа rec.

Доступ к выделенным областям осуществляется с помощью операции разадресации:

```
p1^: = 2; p2^= 4; p3^d:= p1^; p3^.s:= 'Вася';
```

В этих операторах в выделенную память заносятся значения. Динамические переменные можно использовать в операциях, допустимых для величин соответствующего типа, например:

```
inc (p1^); p2^= p1^ + p3^.d;
with p3^ do writeln (d, s);
```

Для освобождения динамической памяти используются процедуры Dispose и Freemem, причем, если память выделялась с помощью new, следует применять Dispose, в противном случае — Freemem.

Процедура Dispose(var p : pointer) освобождает участок памяти, выделенный для размещения динамической переменной процедурой или функцией new, и значение указателя p становится неопределенным.

Процедура Freemem(var p : pointer; size : word) освобождает участок памяти размером size начиная с адреса, находящегося в p. Значение указателя становится неопределенным.

Если требуется освободить память из-под нескольких переменных одновременно, можно применять процедуры Mark и Release.

Процедура Mark(var p : pointer) записывает в указатель p адрес начала участка свободной динамической памяти на момент ее вызова.

Процедура Release(var p : pointer) освобождает участок динамической памяти начиная с адреса, записанного в указатель p процедурой Mark.

При завершении программы используемая ею динамическая память освобождается автоматически.

При работе с динамической памятью часто применяются *вспомогательные функции* Maxavail, Memavail и Sizeof.

Динамические структуры данных

Динамическая структура данных размещается в динамической памяти, ее размер изменяется во время выполнения программы: память выделяется отдельными блоками по мере необходимости. Блоки связываются друг с другом с помощью указателей.

Виды динамических структур (линейные списки, стеки, очереди, деревья) различаются способами связи отдельных элементов и допустимыми операциями. Динамическая структура в отличие от массива или записи может занимать несмежные участки оперативной памяти. Динамические структуры широко применяют и для более эффективной работы с данными, размер которых известен, особенно для решения задач сортировки.

Элемент любой динамической структуры состоит из двух частей: *информационной*, ради хранения которой и создается структура, и *указателей*, обеспечивающих связь элементов друг с другом.

Элемент динамической структуры описывается в виде записи, например:

```
type
pnode = ^node;
node = record
  d: word;   {информационная}
  s: string; {часть}
  p: pnode;  {указатель на следующий элемент}
end;
```

Примечание

Обратите внимание, что тип указателя pnode на запись node определен раньше, чем сама запись. Это не противоречит принципу «использование только после описания», поскольку для описания переменной типа pnode информации вполне достаточно.

Рассмотрим принципы работы с основными динамическими структурами.

Линейные списки

Для работы со списком в программе требуется определить указатель на его начало. Чтобы упростить добавление новых элементов в конец списка, можно также завести указатель на конец списка.

Рассмотрим программу, которая формирует односвязный список из пяти элементов, содержащих число и его текстовое представление, а затем выполняет вставку и удаление заданного элемента. В качестве ключа используется число:

```
program linked_list;
const n = 5;
type pnode = ^node;
  node = record          {элемент списка}
    d : word;
    s : string;
    p : pnode;
  end;
var beg : pnode;         {указатель на начало списка}
      i, key : word;
      s      : string;
      option : word;
const text: array [1 .. n] of string =
  ('one', 'two', 'three', 'four', 'five');

{----добавление элемента в конец списка----}
procedure add(var beg : pnode; d : word; const s : string);
var p : pnode;         {указатель на создаваемый элемент}
  t : pnode;           {указатель для просмотра списка}
begin
  new(p);                      {создание элемента}
  p^.d := d; p^.s := s;        {заполнение элемента}
  p^.p := nil;
  if beg = nil then beg := p {список был пуст}
  else begin                   {список не пуст}
    t := beg;
    while t^.p <> nil do       {проход по списку до конца}
      t := t^.p;
    t^.p := p;  {привязка нового элемента к последнему}
  end
end;

{----поиск элемента по ключу----}
function find(beg: pnode; key: word; var p, pp: pnode) :
boolean;
begin
  p := beg;
  while p <> nil do begin                {1}
    if p^.d = key then begin             {2}
      find := true; exit end;
    pp := p;                             {3}
    p := p^.p;                           {4}
  end;
  find := false;
end;

{----вставка элемента----}
procedure insert(beg : pnode; key, d : word; const s : string);
```

```
var  p   : pnode;     {указатель на создаваемый элемент}
  pkey : pnode;       {указатель на искомый элемент}
  pp   : pnode;       {указатель на предыдущий элемент}
begin
  if not find(beg, key, pkey, pp) then begin
    writeln(' вставка не выполнена'); exit; end;
  new(p);                        {1}
  p^.d := d; p^.s := s;          {2}
  p^.p := pkey^.p;               {3}
  pkey^.p := p;                  {4}
end;

{----удаление элемента----}
procedure del(var beg : pnode; key : word);
var p : pnode;      {указатель на удаляемый элемент}
  pp : pnode;       {указатель на предыдущий элемент}
begin
  if not find(beg, key, p, pp) then begin
    writeln(' удаление не выполнено'); exit; end;
  if p = beg then beg := beg^.p {удаление первого элемента}
  else pp^.p := p^.p;
  dispose(p);
end;

{----вывод списка----}
procedure print(beg : pnode);
var p : pnode;       {указатель для просмотра списка}
begin
  p := beg;
  while p <> nil do begin     {цикл по списку}
    writeln(p^.d:3, p^.s);    {вывод элемента}
    p := p^.p  { переход к следующему элементу списка }
  end;
end;

{----главная программа----}
begin
  for i := 1 to 5 do add(beg, i, text[i]);
  while true do begin
    writeln('1 – вставка, 2 – удаление, 3 – вывод, 4 – выход');
    readln(option);
    case option of
      1: begin                   {вставка}
        writeln('Ключ для вставки?');
        readln(key);
        writeln('Вставляемый элемент?');
        readln(i); readln(s);
        insert(beg, key, i, s);
      end;
      2: begin                   {удаление}
        writeln('Ключ для удаления?');
        readln(key);
        del(beg, key);
```

```
      end;
      3: begin                    {вывод}
        writeln('Вывод списка:');
        print(beg);
      end;
      4: exit;                    {выход}
    end
    writeln;
  end
end.
```

Функция поиска элемента find возвращает true, если искомый элемент найден, и false в противном случае. Поскольку одного факта отыскания элемента недостаточно, функция также возвращает через список параметров два указателя: на найденный элемент p и на предшествующий ему pp. Последний требуется при удалении элемента из списка.

Если элемента с заданным ключом в списке нет, цикл просмотра списка завершится естественным образом, поскольку последний элемент списка содержит nil в поле p указателя на следующий элемент.

Вставка элемента выполняется после элемента с заданным ключом (процедура insert). Под новый элемент выделяется место в динамической памяти (оператор {1}), и информационные поля элемента заполняются переданными в процедуру значениями (оператор {2}). Новый элемент p вставляется между элементами pkey и следующим за ним (его адрес хранится в pkey^.p). Для этого в операторах {3} и {4} устанавливаются две связи (рис. 5.3).

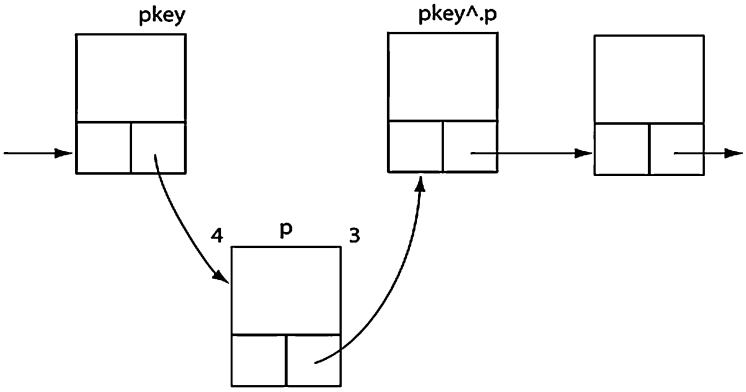

Рис. 5.3. **Вставка элемента в список**

Бинарные деревья

Дерево является рекурсивной структурой данных, поскольку каждое поддерево также является деревом. Действия с такими структурами изящнее всего описываются с помощью рекурсивных

алгоритмов. Например, процедуру обхода всех узлов дерева можно в общем виде описать так:

```
procedure print_tree (дерево);
begin
  print_tree (левое_поддерево)
  посещение корня
  print_tree (правое_поддерево)
end;
```

Приведенная функция позволяет получить последовательность ключей, отсортированную по возрастанию:

```
1, 6, 8, 10, 20, 21, 25, 30
```

Таким образом, деревья поиска можно применять для сортировки значений. При обходе дерева узлы не удаляются.

Тема 6
ПАРАДИГМЫ И ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

6.1. Парадигмы программирования

Парадигма — способ организации программы, т. е. принцип ее построения. Наиболее распространенными являются процедурная и объектно-ориентированная парадигмы. Они различаются способом декомпозиции, положенным в основу при создании программы.

Процедурная декомпозиция состоит в том, что задача, реализуемая программой, делится на подзадачи, а они в свою очередь — на более мелкие этапы, т. е. выполняется пошаговая детализация алгоритма решения задачи.

Объектно-ориентированная декомпозиция предполагает разбиение предметной области на объекты и реализацию этих объектов и их взаимосвязей в виде программы. Кроме того, существуют функциональная и логическая парадигмы, которые кратко рассмотрены в теме 2.

Процедурная парадигма

Процедура парадигма была хронологически первой и долгое время превалировала. В настоящее время она постепенно уступает свое место объектно-ориентированной парадигме, хотя все еще занимает порядка половины рынка разработки программного обеспечения. Она применяется на всех уровнях разработки программного обеспечения.

Парадигма процедурного программирования основана на представлении об управлении машиной со стороны программы (набора процедур). Иными словами, есть четкий набор инструкций, объединенных в процедуры, и четкая последовательность их выполнения. В процессе выполнения процедуры запрашивают необходимые им данные. Таким образом, данные являются подчиненным элементом, а инструкции — главным.

Выполнение процедур естественно назвать процессом, причем из одного процесса может активизироваться другой, что является главным способом управления общим процессом. В момент акти-

визации может производиться передача данных, которые носят в таком случае название параметров. Окончание выполнения процедуры связано с возвращением результата и функции управления в вызывающую программу.

Объектно-ориентированная парадигма

При создании программ на основе объектно-ориентированной парадигмы из предметной области выделяются объекты, поведение и взаимодействие которых моделируются с помощью программы. Кроме того, в программе есть служебные объекты, служащие для хранения объектов, их визуализации и других необходимых функций.

Объект в программе представляется как совокупность данных, характеризующих его состояние, и процедур их обработки, моделирующих его поведение. Вызов процедуры часто называют *посылкой сообщения объекту.*

Выполнение программы состоит в том, что объекты обмениваются *сообщениями.* Это позволяет использовать при программировании понятия, более адекватно отражающие предметную область.

Важным свойством объекта является его обособленность. Детали реализации объекта, т. е. внутренние структуры данных и алгоритмы их обработки, скрыты от пользователя и недоступны для непреднамеренных изменений. Скрытие деталей реализации называется *инкапсуляцией.* Таким образом, объект является «черным ящиком», замкнутым по отношению к внешнему миру. Это позволяет представить программу в более укрупненном виде — на уровне объектов и их взаимосвязей, а следовательно, управлять большим объемом информации и успешно отлаживать более сложные программы.

Инкапсуляция позволяет изменить реализацию объекта без модификации основной части программы, если его интерфейс остался прежним. Простота модификации является очень важным критерием качества программы.

Кроме того, инкапсуляция позволяет использовать объект в другом окружении и быть уверенным, что он не испортит не принадлежащие ему области памяти, а также создавать библиотеки объектов для применения во многих программах.

Каждый год в мире пишется множество новых программ, и важнейшее значение приобретает возможность повторного использования кода. *Преимущество* ООП состоит в том, что для любого объекта можно определить наследников, корректирующих или дополняющих его поведение. При этом нет необходимости иметь доступ к исходному коду родительского объекта.

Наследование позволяет создавать иерархии объектов. Иерархия представляется в виде дерева, в котором более общие объекты располагаются ближе к корню, а более специализированные — на ветвях и листьях. Наследование облегчает использование библиотек

объектов, поскольку программист может взять за основу объекты, созданные кем-то другим, и создать наследников с требуемыми свойствами.

Применение ООП позволяет писать гибкие, расширяемые, хорошо читаемые программы. Во многом это обеспечивается благодаря *полиморфизму*, под которым понимается возможность во время выполнения программы с помощью одного и того же имени выполнять разные действия или обращаться к объектам разного типа. Чаще всего понятие полиморфизма связывают с механизмом виртуальных методов.

6.2. Понятие программного продукта

Основное требование, предъявляемое в настоящее время к программе, — надежность. Под *надежностью* подразумевается способность программы работать в полном соответствии со спецификацией и адекватно реагировать на любые действия пользователя.

Программа должна также обладать *расширяемостью*, т. е. допускать оперативное внесение необходимых изменений и дополнений. Может показаться странным, зачем с самого начала думать о будущих изменениях программы, но ведь для любого сколько-нибудь удачного коммерческого продукта выход новых версий — единственный способ не потерять популярность среди пользователей.

Кроме того, программа должна быть *выпущена к заявленному сроку*. Это значит, что весь процесс производства программы должен четко планироваться и контролироваться.

В результате быстрого развития отрасли и жесткой конкуренции отходят на второй план такие критерии качества программы, как *эффективность* и требуемые *ресурсы*, например объем внешней и оперативной памяти. Однако это не означает, что этим критериям вообще не следует уделять внимание.

Рассматривая характеристики качества более подробно, известный специалист Стив Макконнелл разделяет их на внешние,

значимые для пользователя программного обеспечения, и внутренние, важные для разработчика.

Внешними характеристиками качества программного обеспечения являются:

— корректность (наличие/отсутствие дефектов в спецификации, проекте и реализации);

— практичность (легкость изучения и использования);

— эффективность (степень использования системных ресурсов);

— надежность (способность системы выполнять необходимые функции; интервал между отказами);

— целостность (способность предотвращать неавторизованный или некорректный доступ);

— адаптируемость (возможность использования в других областях и средах);

— правильность (степень безошибочности данных, выдаваемых системой);

— живучесть (способность продолжать работу при недопустимых данных или в напряженных условиях).

К *внутренним характеристикам* качества относятся такие, как удобство сопровождения, тестируемость, удобочитаемость, гибкость, портируемость, возможность повторного использования и понятность.

Между многими из характеристик существуют сложные зависимости: улучшение одних аспектов ведет к ухудшению других и улучшению третьих.

6.3. Обзор современных технологий разработки программного обеспечения. Понятие о UML

Технология программирования — способ организации процесса создания программы. За более чем полувековую историю развития программирования был разработан целый ряд технологий, каждая из которых в чем-то усовершенствовала предыдущие. Выбор технологии зависит от сложности программного обеспечения, наличия готовых компонентов, имеющихся ресурсов и т. д.

Все технологии можно условно разбить на два семейства: *тяжеловесные* (*heavyweight*) — применяются при фиксированных требованиях и многочисленной группе разработчиков разной квалификации, и *облегченные* (*lightweight, agile*) — применяются при малочисленной группе квалифицированных разработчиков и грамотном заказчике, который имеет возможность участвовать в процессе. Любая технология имеет целью создание качественного программного продукта.

Нисходящее и восходящее проектирование

Одна из основных идей, положенных в большинство известных технологий программирования, — **нисходящее проектирование** (*Top-Down Programming* — программирование «сверху вниз»). Существуют также другие названия: «метод пошаговой детализации», «систематическое программирование», «иерархическое программирование». Его принцип состоит в том, что сначала определяются основные функции, которые должны быть обеспечены разрабатываемой программой, а затем они конкретизируются с помощью набора дополнительных функций. Например, основная функция —

обработка файла. Детализирующие функции — открыть файл, обработать все записи, закрыть файл.

В методе нисходящего проектирования сначала пишется основная программа, используя средства вызова подпрограмм, причем в качестве подпрограмм вначале вводятся «заглушки» вида: «Вызвали подпрограмму номер…». Затем, будучи уверенным в правильности логического построения основной программы, пишется каждая подпрограмма, вызывая по мере необходимости подпрограммы более низкого уровня. Этот последовательный процесс продолжается, пока программа не будет завершена и проверена.

При другом методе, **восходящем проектировании** (программировании «снизу вверх»), сначала пишутся подпрограммы нижнего уровня, тщательно тестируются и отлаживаются. Далее добавляют подпрограммы более высокого уровня, которые вызывают подпрограммы нижнего уровня, и так до тех пор пока не будет достигнута программа самого верхнего уровня. Метод проектирования «снизу вверх» пригоден при наличии больших библиотек стандартных подпрограмм.

Иногда лучшим является гибрид двух методов. Однако в обоих случаях каждая подпрограмма должна быть небольшой, так чтобы можно было охватить одним взглядом всю ее логику.

Классическая модель жизненного цикла программного обеспечения

Первой из применяющихся в настоящее время технологий был структурный подход: его основные принципы были разработаны в 1960-х гг. и послужили основой для других, более современных методологий. В настоящее время его успешно применяют в небольших проектах и как составную часть других технологий.

Структурное программирование — это способ создания программ, позволяющий путем соблюдения определенных правил уменьшить время разработки и облегчить возможность модификации программы. Технология структурного программирования охватывает все этапы разработки программы: спецификацию, проектирование, собственно программирование и тестирование. Технология структурного программирования также известна как классическая (водопадная, каскадная) модель жизненного цикла программного обеспечения (рис. 6.1).

Как следует из названия, технология структурного программирования позволяет создавать программы, имеющие простую структуру. В основе структурного подхода лежит рассмотренная выше идея *нисходящего проектирования*.

При написании структурной программы применяется рассмотренный в теме 4 фиксированный набор конструкций, называемых *базовыми*, что позволяет уменьшить как количество ошибок в про-

грамме, так и их цену. Под *ценой ошибки* понимается стоимость ее исправления: она тем выше, чем позже в процессе разработки обнаружена ошибка.

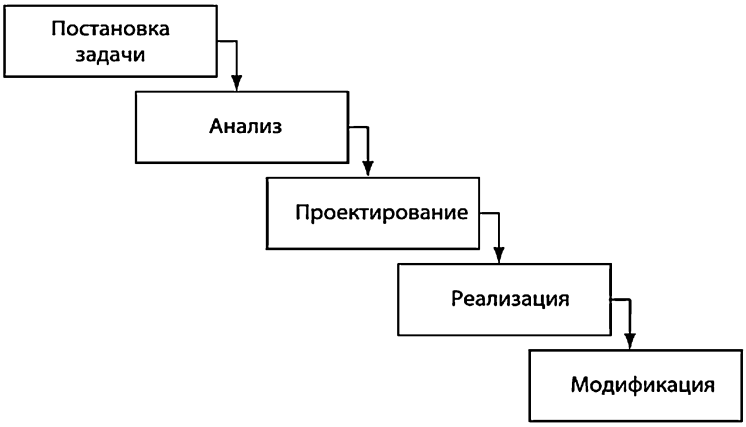

Рис. 6.1. **Классическая модель жизненного цикла разработки программного обеспечения**

Этапы создания структурной программы

Постановка задачи. Создание любой программы начинается с постановки задачи. Изначально задача формулируется в терминах предметной области, и необходимо перевести ее на язык понятий, более близких к программированию. Поскольку программист редко досконально разбирается в предметной области, а заказчик — в программировании (простой пример: требуется написать бухгалтерскую программу), постановка задачи может стать весьма непростым итерационным процессом. Кроме того, при постановке задачи заказчик зачастую не может четко и полно сформулировать свои требования и критерии.

На этом этапе также определяется среда, в которой будет выполняться программа: требования к аппаратуре, используемая ОС и другое программное обеспечение.

Постановка задачи завершается созданием *технического задания*, а затем *внешней спецификации программы*, включающей в себя:

— описание исходных данных и результатов (типы, форматы, точность, способ передачи, ограничения)[1];

— описание задачи, реализуемой программой;

— способ обращения к программе;

— описание возможных аварийных ситуаций и ошибок пользователя.

[1] Под типами и форматами не имеются в виду типы языка программирования.

Таким образом, программа рассматривается как «черный ящик», для которого определена функция и входные и выходные данные.

Выбор модели и метода решения задачи. На этом этапе анализируются условия задачи, и на этом основании строится модель задачи и определяется общий метод ее решения. При построении модели выделяются характеристики задачи, существенные с точки зрения рассмотрения, т. е. выполняется ее абстрагирование. Эти характеристики должны представляться в модели с необходимой полнотой и точностью. Иными словами, на этом этапе постановка задачи формализуется и на этой основе определяется общий метод ее решения. При наличии нескольких методов наилучший выбирается исходя из критериев сложности, эффективности, точности в зависимости от конкретных задач, стоящих перед программистом.

Разработка внутренних структур данных. Большинство алгоритмов зависят от того, каким образом организованы данные, поэтому интуитивно ясно, что начинать проектирование программы надо не с алгоритмов, а с разработки структур, необходимых для представления входных, выходных и промежуточных данных. При этом принимаются во внимание многие факторы, например ограничения на размер данных, необходимая точность, требования к быстродействию программы. Структуры данных могут быть статическими или динамическими.

При решении вопроса о том, как будут организованы данные в программе, полезно задать себе следующие вопросы.

— Какая точность представления данных необходима?
— В каком диапазоне лежат значения данных?
— Ограничено ли максимальное количество данных?
— Обязательно ли хранить их в программе одновременно?
— Какие действия потребуется выполнять над данными?

Например, если максимальное количество однотипных данных, которые требуется обработать, известно и невелико, проще всего завести для их хранения статический массив. Если таких массивов много, сегментов данных и стека может оказаться недостаточно, и придется отвести под эти массивы место в динамической памяти.

Если максимальное количество данных неизвестно и постоянно изменяется во время работы программы, то для их хранения используют динамические структуры. Выбор вида структуры зависит от требуемых операций над данными. Например, для быстрого поиска элементов лучше всего подходит бинарное дерево, а если данные требуется обрабатывать в порядке поступления, применяется очередь.

Проектирование. Под *проектированием программы* понимается определение общей структуры и взаимодействия модулей. На этом этапе применяется *технология нисходящего проектирования*, ос-

новная идея которого рассмотрена выше. При этом используется метод пошаговой детализации.

Можно представить себе этот процесс так, что сначала программа пишется на языке некоторой гипотетической машины, которая способна понимать самые обобщенные действия, а затем каждое из них описывается на более низком уровне абстракции и т. д. Очень важной на этом этапе является *спецификация интерфейсов*, т. е. определение способов взаимодействия подзадач.

Для каждой подзадачи составляется внешняя спецификация, аналогичная приведенной ранее. На этом же этапе решаются вопросы разбиения программы на модули, главным критерием при этом является минимизация их взаимодействия. Одна задача может реализовываться с помощью нескольких модулей, и наоборот, в одном модуле может решаться несколько задач. На более низкий уровень проектирования переходят только после окончания проектирования верхнего уровня. Алгоритмы записывают в обобщенной форме, например словесной, в виде обобщенных блок-схем или другими способами.

ВНИМАНИЕ

На этапе проектирования следует учитывать возможность будущих модификаций программы и стремиться проектировать программу таким образом, чтобы вносить изменения было как можно проще.

Поскольку неизвестно, какие изменения придется выполнить, это пожелание напоминает создание «общей теории всего»; на практике надо ограничиться разумными компромиссами. Программист исходя из своего опыта и здравого смысла решает, какие именно свойства программы может потребоваться изменить или усовершенствовать в будущем.

Процесс проектирования является итерационным, поскольку в программах реального размера невозможно продумать все детали с первого раза.

Структурное программирование. Программирование здесь рассматривается «в узком смысле», т. е. понимается как запись программы на языке программирования по готовому алгоритму. Этот процесс часто называют *кодированием*, чтобы отличить его от полного цикла разработки программы.

Кодирование также организуется по принципу «сверху вниз»: вначале кодируются модули самого верхнего уровня и составляются тестовые примеры для их отладки. При этом на месте еще не написанных модулей следующего уровня ставятся заглушки, которые в простейшем случае просто выдают сообщение о том, что им передано управление, а затем возвращают его в вызывающий модуль.

В других случаях заглушка может выдавать значения, заданные заранее или вычисленные по упрощенному алгоритму.

Таким образом, сначала создается логический скелет программы, который затем обрастает плотью кода. Казалось бы, более логично применять к процессу программирования восходящую технологию: написать и отладить сначала модули нижнего уровня, а затем объединять их в более крупные фрагменты, но этот подход имеет ряд недостатков.

Во-первых, в процессе кодирования верхнего уровня могут быть вскрыты те или иные трудности проектирования более низких уровней программы (просто потому, что при написании программы ее логика продумывается более тщательно, чем при проектировании). Если подобная ошибка обнаруживается в последнюю очередь, требуются дополнительные затраты на переделку уже готовых модулей нижнего уровня.

Во-вторых, для отладки каждого модуля, а затем более крупных фрагментов программы требуется каждый раз составлять свои тестовые примеры, и программист часто вынужден имитировать то окружение, в котором должен работать модуль. Нисходящая же технология программирования обеспечивает естественный порядок создания тестов — возможность нисходящей отладки, которая рассмотрена далее.

Этапы проектирования и программирования совмещены во времени: в идеале сначала проектируется и кодируется верхний уровень, затем — следующий и т. д. Такая стратегия применяется потому, что она снижает цену ошибки, поскольку в процессе кодирования может возникнуть необходимость внести изменения, отражающиеся на модулях нижнего уровня.

Нисходящее тестирование. Этот этап записан последним, но это не значит, что тестирование не должно проводиться на предыдущих этапах. И проектирование, и программирование обязательно должны сопровождаться написанием *набора тестов* — проверочных исходных данных и соответствующих им наборов эталонных реакций.

Необходимо различать процессы тестирования и отладки программы. **Тестирование** — это процесс, посредством которого проверяется правильность программы. Тестирование носит позитивный характер, его цель — показать, что программа работает правильно и удовлетворяет всем проектным спецификациям. **Отладка** — процесс исправления ошибок в программе, при котором цель исправить все ошибки не ставится. Исправляют ошибки, обнаруженные при тестировании. При планировании следует учитывать, что процесс обнаружения ошибок подчиняется закону насыщения, т. е. большинство ошибок обнаруживаются на ранних стадиях тестирования, и чем меньше в программе осталось ошибок, тем дольше искать каждую из них.

Существует две стратегии тестирования: «черный ящик» и «белый ящик». При использовании первой внутренняя структура программы во внимание не принимается и тесты составляются так, чтобы полностью проверить функционирование программы на корректных и некорректных входных воздействиях.

Стратегия «белого ящика» предполагает проверку всех ветвей алгоритма. Общее число ветвей определяется комбинацией всех альтернатив на каждом этапе. Это конечное число, но оно может быть очень большим, поэтому программа разбивается на фрагменты, после исчерпывающего тестирования которых они рассматриваются как элементарные узлы более длинных ветвей. Кроме данных, обеспечивающих выполнение операторов в требуемой последовательности, тесты должны содержать *проверку граничных условий* (например, переход по условию $x > 10$ должен проверяться для значений, больших, меньших и равных 10). Отдельно проверяется реакция программы на *ошибочные исходные данные.*

Недостатком стратегии «белого ящика» является то, что обнаружить с ее помощью отсутствующую ветвь невозможно, а стратегия «черного ящика» требует большого числа вариантов входных воздействий, поэтому на практике применяют сочетание обеих стратегий.

ВНИМАНИЕ

Идея нисходящего тестирования предполагает, что к тестированию программы приступают еще до того, как завершено ее проектирование. Это позволяет раньше опробовать основные межмодульные интерфейсы, а также убедиться в том, что программа в основном удовлетворяет требованиям пользователя. Только после того как логическое ядро испытано настолько, что появляется уверенность в правильности реализации основных интерфейсов, приступают к кодированию и тестированию следующего уровня программы.

Естественно, полное тестирование программы, пока она представлена в виде скелета, невозможно, однако добавление каждого следующего уровня позволяет постепенно расширять область тестирования.

Этап комплексной отладки на уровне системы при нисходящем проектировании занимает меньше времени, чем при восходящем, и приносит меньше сюрпризов, поскольку вероятность появления серьезных ошибок, затрагивающих большую часть системы, гораздо ниже. Кроме того, для каждого подключаемого к системе модуля уже создано его окружение, и выходные данные отлаженных модулей можно использовать как входные для тестирования других, что облегчает процесс тестирования. Это не значит, что модуль надо подключать к системе совсем «сырым», бывает удобно провести часть тестирования автономно, поскольку сгенерировать на входе

системы все варианты, необходимые для тестирования отдельного модуля, трудно.

Каскадная модель в чистом виде применяется только для несложных программ, поскольку трудно заранее предусмотреть все детали реализации. Более реальной является схема с промежуточным контролем (рис. 6.2). Контроль, выполняемый после каждого этапа, позволяет при необходимости вернуться на любой вышележащий уровень и внести необходимые изменения.

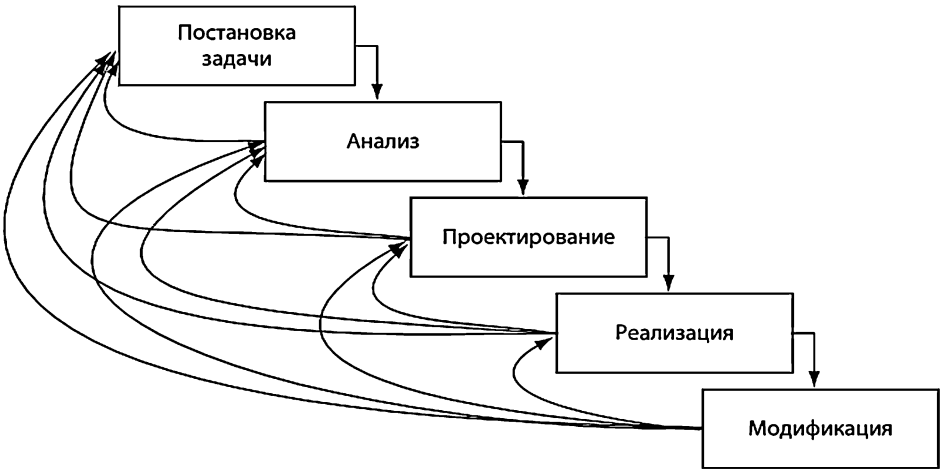

Рис. 6.2. **Модель разработки программного обеспечения с промежуточным контролем**

Основная опасность использования такой схемы связана с тем, что разработка никогда не будет завершена, постоянно находясь в состоянии уточнения и усовершенствования.

Спиральная модель разработки

В середине 1980-х гг. была предложена спиральная схема (рис. 6.3). В соответствии с ней программное обеспечение создается итерационно с использованием *метода прототипирования*, основанного на создании прототипов. Появление прототипирования привело к тому, что процесс модификации программного обеспечения стал восприниматься не как помеха, а как необходимый этап разработки.

Прототипом называют действующий программный продукт, реализующий отдельные функции и внешние интерфейсы разрабатываемого программного обеспечения.

На первой итерации, как правило, специфицируют, проектируют, реализуют и тестируют интерфейс пользователя. На второй — добавляют некоторый ограниченный набор функций. На после-

дующих этапах этот набор расширяют, наращивая возможности продукта.

Основным достоинством спиральной схемы является то, что начиная с некоторой итерации, на которой обеспечена определенная функциональная полнота, продукт можно предоставлять пользователю, что позволяет:

— сократить время до появления первых версий программного продукта;

— заинтересовать большое количество пользователей, обеспечивая быстрое продвижение следующих версий продукта на рынке;

— ускорить формирование и уточнение спецификаций за счет появления практики использования продукта;

— уменьшить вероятность морального устаревания системы за время разработки.

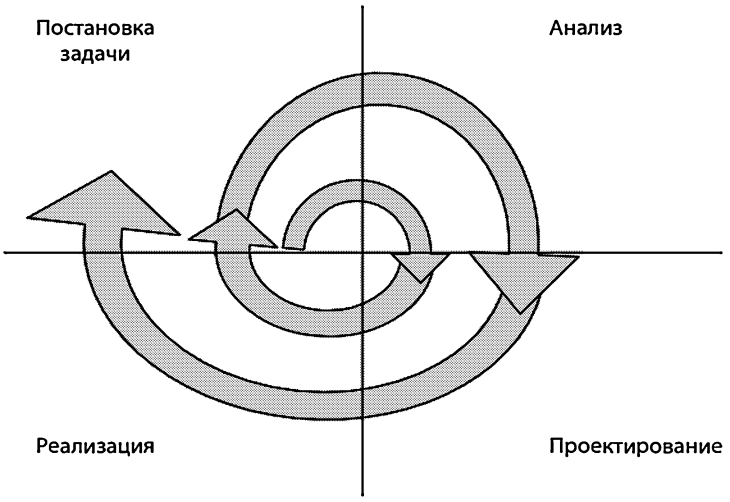

Рис. 6.3. **Спиральная модель разработки программного обеспечения**

Основной проблемой использования спиральной схемы является определение моментов перехода на следующие стадии. Для ее решения обычно ограничивают сроки прохождения каждой стадии, основываясь на экспертных оценках.

Отличительной особенностью современного этапа развития технологии программирования является создание и внедрение автоматизированных технологий разработки и сопровождения программного обеспечения, которые называют CASE-технологиями (*Computer Aided Software/System Engineering* — разработка программного обеспечения/программных систем с использованием компьютерной поддержки).

В настоящее время использование CASE-средств является основным способом создания коммерческих программных продуктов.

Эти средства автоматизируют трудоемкие и требующие особого внимания операции — например, сборку программы из отдельных модулей, контроль внесения изменений в проект, текстирование, ведение документации, частично даже формирование программного кода. Это повышают производительность труда программистов и улучшает качество программного продукта.

Недостатками CASE-средств является их высокая стоимость и сложность освоения разработчиками. Таким образом, их имеет смысл использовать в сложных проектах.

Экстремальное программирование

Основная идея экстремального программирования (XP) — устранить высокую стоимость изменений, вносимых в программное обеспечение в процессе как разработки, так и эксплуатации. Цикл разработки в XP состоит из очень коротких итераций. Четырьмя базовыми действиями в цикле являются выслушивание заказчика, проектирование, кодирование и тестирование. Заказчик постоянно присутствует в группе разработчиков. При принятии решений всегда стремятся выбрать самое простое, тесты пишутся еще до написания кода, а сборка системы выполняется ежедневно.

Большинство принципов, применяемых в XP, диктуется здравым смыслом и применяется в том или ином виде и в других технологиях, просто в XP они достигают «экстремальных» значений.

Унифицированный процесс (RUP)

Разработчиками IBM Rational Unified Process (RUP) являются Г. Буч, А. Якобсон, Д. Рамбо (компания Rational, 1998).

IBM Rational Unified Process — это:

— подход к разработке программных систем (ПС), основанный на использовании лучших практических методов, успешно зарекомендовавших себя во многих проектах разработки ПС;

— четко определенный процесс (технологическая процедура), описывающий структуру жизненного цикла проекта, роли и ответственности отдельных исполнителей, выполняемые ими задачи и используемые в процессе разработки модели, отчеты и т. д.;

— готовый продукт, предоставляемый в виде веб-сайта, содержащего все необходимые модели и документы с описанием процесса.

RUP предлагает разработчикам не жесткие правила, регламентирующие выполнение всех действий в ходе разработки, а *набор достаточно гибких методов и подходов*, из которых разработчик может выбирать то, что более всего соответствует его задачам и особенностям проекта.

Таким образом, RUP является обобщенным каркасом процесса разработки программного обеспечения. Он может быть специали-

зирован для широкого круга программных систем, уровней компетенции, областей применения и размеров проекта.

RUP управляет действиями всех его участников: разработчиков, руководства, пользователей и заказчиков. Процесс определяет, кто, когда и что делает и как достичь определенной цели. Хороший процесс предоставляет указания по эффективной разработке качественного программного продукта. В результате он уменьшает риск и повышает предсказуемость.

RUP — итерационный процесс. Создавать современные сложные программные системы последовательно, т. е. сначала определять все проблемы, затем принимать все проектные решения, разрабатывать и, наконец, проверять изделие, невозможно. Такой подход, рассмотренный выше в разделе «Классическая модель жизненного цикла программного обеспечения», в современной информационной индустрии не оправдывает себя, поскольку его использование часто приводит к непредсказуемому увеличению проектной стоимости и сроков выпуска ПС. Эффективной альтернативой «водопадной модели» служит итерационный процесс разработки ПС.

При итерационном процессе каждая из фаз процесса разработки ПС состоит из итераций, целью которых является последовательное осмысление стоящих проблем, наращивание эффективности решений и снижение риска потенциальных ошибок в проекте. На выходе каждой итерации создается законченная версия работающего программного продукта.

При этом подходе можно гибко учитывать новые требования или производить тактические изменения в деловых целях, он позволяет выявлять проблемы и разрешать их на самых ранних этапах разработки, что связано с меньшими затратами.

Итерация — это законченный цикл разработки, приводящий к выпуску конечного продукта или некоторой его сокращенной версии, которая расширяется от итерации к итерации, чтобы в конце концов стать законченной системой.

RUP использует описание проектируемой системы в виде моделей. Каждая модель описывает систему с определенной точки зрения. Существуют модели вариантов использования, анализа, проектирования, развертывания, реализации, тестирования. Все модели связаны, они полностью описывают систему. Набор моделей дает варианты обозрения системы для всех сотрудников. Для описания моделей используется язык UML.

UML (Unified Modeling Language) является языком для специфицирования, визуализации, конструирования и документирования программных продуктов, а также используется в бизнес-моделировании и моделировании любых иных (не программных) систем. Примеры диаграмм UML приведены на рис. 6.4, 6.5.

Рис. 6.4. **Диаграмма деятельности (Activity diagram)**

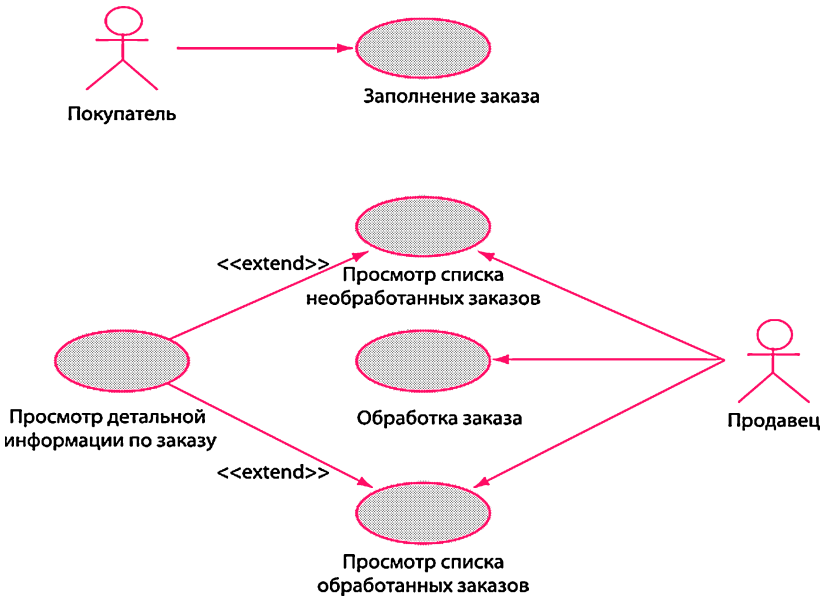

Рис. 6.5. **Диаграмма вариантов использования (Use case diagram)**

6.4. Введение в объектно-ориентированное программирование

Объектно-ориентированное программирование представляет собой дальнейшее развитие идей структурного программирования, основной целью которого является создание программ простой структуры. Это достигается за счет разбиения программы на максимально обособленные части.

Структурная программа состоит из совокупности подпрограмм, связанных с помощью интерфейсов. Подпрограмма работает с данными, которые либо являются локальными, либо передаются ей в качестве параметров. Каждая подпрограмма предназначена для работы с определенными типами данных. Ошибки часто связаны с тем, что в подпрограмму передаются неверные данные.

Естественный путь избежать таких ошибок — связать в одно целое данные и все подпрограммы, которые предназначены для их обработки. Эта идея лежит в основе ООП: из предметной области выделяются объекты, поведение и взаимодействие которых моделируются с помощью программы. Основные особенности объектно-ориентированной парадигмы были кратко описаны в начале этой темы в подтеме 6.1.

Объект, или класс, является абстрактным типом данных, создаваемым программистом. При описании класса определяются его *поля* (данные) и *методы* (подпрограммы их обработки). Конкретные переменные объектного типа называются *экземплярами* объекта. Аналогия из обыденной жизни: собака является объектом, а экземплярами — конкретные Жучка или Терри.

Выполнение программы состоит в том, что объекты обмениваются *сообщениями*. Это позволяет использовать при программировании понятия, более адекватно отражающие предметную область.

К *преимуществам* ООП относятся:

— использование при программировании понятий, более близких к предметной области;

— более простая структура программы в результате инкапсуляции, т. е. объединения свойств и поведения объекта и скрытия деталей его реализации;

— возможность повторного использования кода за счет наследования;

— сравнительно простая возможность модификации программы;

— возможность создания библиотек объектов.

Эти преимущества проявляются при создании программ большого объема и классов программ. Однако создание объектно-ориентированной программы представляет собой весьма непростую задачу, поскольку в процесс добавляется этап разработки иерархии

объектов, а плохо спроектированная иерархия приводит к появлению сложных и запутанных программ.

Кроме того, идеи ООП непросты для понимания и в особенности для практического применения. Чтобы эффективно использовать готовые объекты, необходимо освоить большой объем достаточно сложной информации. Неграмотное же применение ООП способно привести к созданию программ, сочетающих недостатки и структурного, и объектно-ориентированного подхода, и лишенных их преимуществ.

Далее рассматривается конкретная реализация объектной модели в языке ПАСКАЛЬ.

Объекты

В языке ПАСКАЛЬ **объект** — это тип данных, определяемый программистом. Объект похож на тип «запись» (record), но кроме *полей данных* в нем можно описывать методы. *Методами* называются подпрограммы, предназначенные для работы с полями объекта. Внутри объекта описываются только заголовки методов:

```
type имя = object
  [private]
    описание элементов
  [public]
    заголовки методов
end;
```

Поля и методы называются *элементами объекта*. Их видимостью управляют директивы private и public[1]. Ключевое слово private (*закрытые*) ограничивает видимость перечисленных после него элементов файлом, в котором описан объект. Действие директивы распространяется до другой директивы или до конца объекта. В объекте может быть произвольное количество разделов private и public. По умолчанию все элементы объекта считаются видимыми извне, т. е. являются public (открытыми).

Для каждого *поля объекта* задается его имя и тип. Он может быть любым, кроме типа того же объекта, но может быть указателем на этот тип.

При определении состава *методов* исходят из требуемого поведения объекта. Каждое действие, которое должен выполнять объект, оформляется в виде отдельной процедуры или функции.

Пример

Пусть объект моделирует персонаж компьютерной игры, который характеризуется своим положением и изображением на экране, определенными «запасами» здоровья и вооружения и цветом, а также умеет атаковать противника:

[1] Квадратные скобки означают, что эти директивы являются необязательными.

```
type monster = object
procedure init (x_, y_, health_, ammo_: word);
procedure attack;
procedure draw;
procedure erase;
procedure hit;
procedure move (x_, y_: word);
private
  x, y         : word;
  health, ammo : word
  word; color  : word;
end;
```

Вся внешняя информация, необходимая для выполнения действий с объектом, должна передаваться методам в качестве параметров. Параметры могут иметь любой тип, в том числе и тип того же объекта. Имена параметров не должны совпадать с именами полей объекта.

Описание методов (текст подпрограмм) размещается вне объекта в разделе описания процедур и функций:

```
procedure monster.init (x_, y_, health_, ammo_: word);
begin
  x := x_; y := y_;
  health := health_; ammo := ammo_;
  color := yellow;
end;
procedure monster.draw;
begin
  … {процедуры вывода изображения}
end;
```

Поля объекта используются внутри методов непосредственно, без указания имени объекта. Методы представляют собой разновидность подпрограмм, поэтому внутри них можно описывать локальные переменные. Как правило, поля объекта объявляют private, а методы — public.

Объекты часто описывают в модулях. Тип объекта определяется в интерфейсном разделе модуля, а методы объекта — в разделе реализации. Объектный тип можно определять только в разделе описания типов самого внешнего блока программы или модуля.

Экземпляры объектов

Экземпляр объекта — это переменная объектного типа. Время жизни и видимость объектов зависят от вида и места их описания и подчиняются общим правилам языка ПАСКАЛЬ. Экземпляры объектов можно создавать в статической или динамической памяти:

```
var Vasia: monster; {описывается статический объект}
  pm: ^monster;     {описывается указатель на объект}
```

```
...
new (pm);              {создается динамический объект}
```

Можно определять массивы объектов или указателей на объекты и создавать из них динамические структуры данных. Если объектный тип описан в модуле, для создания в программе переменных этого типа следует подключить модуль в разделе uses:

```
uses graph, monsters;
```

Доступ к элементам объекта осуществляется либо с использованием составного имени, либо с помощью оператора with:

```
Vasia.erase;
with pm^ do
begin
  init (100, 100, 30);
  draw;
end;
```

Если объект описан в модуле, то получить доступ к значениям полей со спецификатором private в программе можно только через обращение к соответствующим методам.

При создании каждого объекта выделяется память, достаточная для хранения всех его полей. Методы объекта хранятся в одном экземпляре. Для того чтобы методу было известно, с данными какого экземпляра объекта он работает, при вызове ему в неявном виде передается параметр self, определяющий место расположения данных этого объекта.

Объекты одного типа можно *присваивать* друг другу, при этом выполняется поэлементное копирование всех полей. Кроме того, в языке ПАСКАЛЬ определены правила расширенной совместимости типов объектов. Они рассмотрены далее. Все остальные действия выполняются над отдельными полями объектов.

Иерархии объектов

Управлять большим количеством разрозненных объектов сложно. С этим можно справиться путем упорядочивания и ранжирования объектов, т. е. объединяя общие для нескольких объектов свойства в одном объекте и используя этот объект в качестве базового.

Эту возможность предоставляет механизм *наследования*. Он позволяет строить иерархии, в которых объекты-потомки получают свойства объектов-предков и могут дополнять их или изменять. Таким образом, наследование обеспечивает возможность повторного использования кода.

При описании объекта имя его предка записывается в круглых скобках после ключевого слова object.

Пусть требуется ввести в игру еще один тип персонажей, который должен обладать свойствами объекта monster, но по-другому выглядеть и атаковать:

```
type daemon = object (monster)
  procedure init(x_, y_, health_, ammo_, magic_ : word);
  procedure attack;
  procedure draw;
  procedure erase;
  procedure wizardry;
private
  magic : word;
end;

{----реализация методов объекта daemon----}
procedure daemon.init(x_, y_, health_, ammo_, magic_ : word);
begin
  inherited init(x_, y_, health_, ammo_);
  color := green;
  magic := magic_;
end;
...
```

Наследование полей. Унаследованные поля доступны в объекте точно так же, как и его собственные. *Изменить или удалить поле при наследовании нельзя.* Таким образом, потомок всегда содержит число полей, большее или равное числу полей своего предка.

Наследование методов. В «потомке» объекта можно не только описывать новые методы, но и переопределять существующие. Метод можно переопределить либо полностью, либо дополнив метод «предка».

Последний вариант иллюстрируется с помощью метода init. В объекте daemon этот метод переопределен. Из него с помощью ключевого слова **inherited** (унаследованный) сначала вызывается старый метод, а затем выполняются дополнительные действия. Можно вызвать метод «предка» и явным образом с помощью конструкции monster.init.

Совместимость типов объектов

ПАСКАЛЬ — язык со строгой типизацией. Операнды выражений, параметры подпрограмм и их аргументы должны подчиняться правилам соответствия типов. Для объектов понятие совместимости расширено: *производный тип совместим со своим родительским типом*. Эта расширенная совместимость типов имеет три формы:

1) между экземплярами объектов;
2) между указателями на экземпляры объектов;
3) между параметрами и аргументами подпрограмм.

Во всех трех случаях *родительскому объекту может быть присвоен экземпляр любого из его потомков*, но не наоборот (рис. 6.6).

Например, если определены переменные:

```
type
  pmonster = ^monster;
  pdaemon = ^daemon;
```

```
var m: monster;
  d : daemon;
  pm : pmonster;
  pd : pdaemon;
```

то допустимы приведенные далее операторы присваивания:

```
m : = d;
pm : = pd;
```

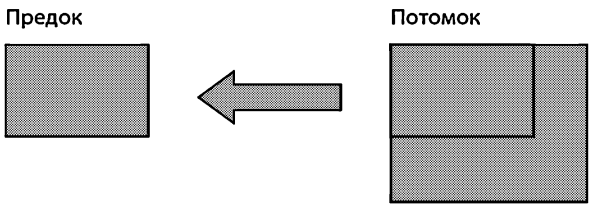

Рис. 6.6. **Направление совместимости типов**

Поля и методы, введенные в потомке, после таких присваиваний недоступны. Даже если метод переопределен в потомке, через указатель на предка вызывается метод, описанный в предке. Так, в результате выполнения оператора pm^.draw на экране появится изображение объекта-предка, потому что тип вызываемого метода соответствует типу указателя, а не типу того объекта, на который он ссылается.

Если известно, что указатель на предка на самом деле хранит ссылку на потомка, можно обратиться к элементам, определенным в потомке, с помощью явного преобразования типа, например pdaemon (pm)^.wizardry.

Если объект является параметром подпрограммы, ему может соответствовать аргумент того же типа или типа любого из его потомков.

Объекты, фактический тип которых может изменяться во время выполнения программы, называются *полиморфными*. Полиморфным может быть объект, определенный через указатель или переданный в подпрограмму по адресу. Полиморфные объекты обычно применяются вместе с виртуальными методами.

Виртуальные методы

Рассмотрим работу компилятора при использовании в программе иерархий объектов. Исполняемые операторы программы в виде инструкций процессора находятся в сегменте кода. Каждая подпрограмма имеет точку входа. При компиляции вызов подпрограммы заменяется последовательностью команд, которая передает управление в эту точку. Этот процесс называется р*азрешением ссылок*, или *ранним связыванием*, потому что адреса перехода на подпрограммы компилятор вставляет до выполнения программы.

Ясно, что с помощью этого механизма не удастся обеспечить возможность вызова методов разных объектов (предков и потомков) с помощью одного и того же указателя или из одной и той же подпрограммы. Это можно сделать только в том случае, если ссылки будут разрешаться на этапе выполнения программы в момент вызова метода. Такой механизм называется *поздним связыванием* и реализуется с помощью виртуальных методов. Они определяются с помощью директивы virtual, которая записывается в заголовке метода:

```
procedure attack; virtual;
```

Объявление метода виртуальным означает, что все ссылки на этот метод будут разрешаться по факту его вызова, т. е. не на стадии компиляции, а во время выполнения программы. Для реализации этой возможности необходимо, чтобы адреса виртуальных методов хранились там, где ими можно в любой момент воспользоваться, поэтому компилятор формирует для них таблицу виртуальных методов (Virtual Method Table — VMT). Для каждого объектного типа создается одна VMT.

Каждый объект во время выполнения программы должен иметь доступ к VMT. Эта связь устанавливается при создании объекта с помощью специального метода, называемого *конструктором*. Класс, имеющий хотя бы один виртуальный метод, должен содержать конструктор, например:

```
type monster = object
constructor init (x_, y_, health_, ammo_: word);
  procedure attack; virtual;
  procedure draw;   virtual;
  procedure erase;  virtual;
  procedure hit;
  procedure move (x_, y_: word);
private
  x, y : word;
  health, ammo : word;
  color : word; end;
daemon = object (monster)
  constructor init (x_, y_, health_, ammo_, magic_: word);
  procedure attack; virtual;
  procedure draw;   virtual;
  procedure erase;  virtual;
  procedure wizardry;
private
  magic: word;
end;
```

По ключевому слову constructor компилятор вставляет в начало метода фрагмент, который записывает ссылку на VMT в специальное поле объекта. Прежде чем использовать виртуальные методы, необходимо вызвать конструктор объекта.

Конструктор обычно используется для инициализации объекта. В нем выполняется выделение памяти под динамические переменные или структуры, если они есть в объекте, и присваиваются начальные значения полям объекта. Поскольку связь с VMT устанавливается в самом начале конструктора, в его теле также можно пользоваться виртуальными методами.

Объект может содержать несколько конструкторов. Конструктор надо вызывать явным образом для каждого создаваемого объекта.

Вызов виртуального метода выполняется так: из объекта берется адрес его VMT, из VMT выбирается адрес метода, а затем управление передается этому методу (рис. 6.7). Таким образом, при использовании виртуальных методов из всех одноименных методов иерархии всегда выбирается тот, который соответствует фактическому типу вызвавшего его объекта.

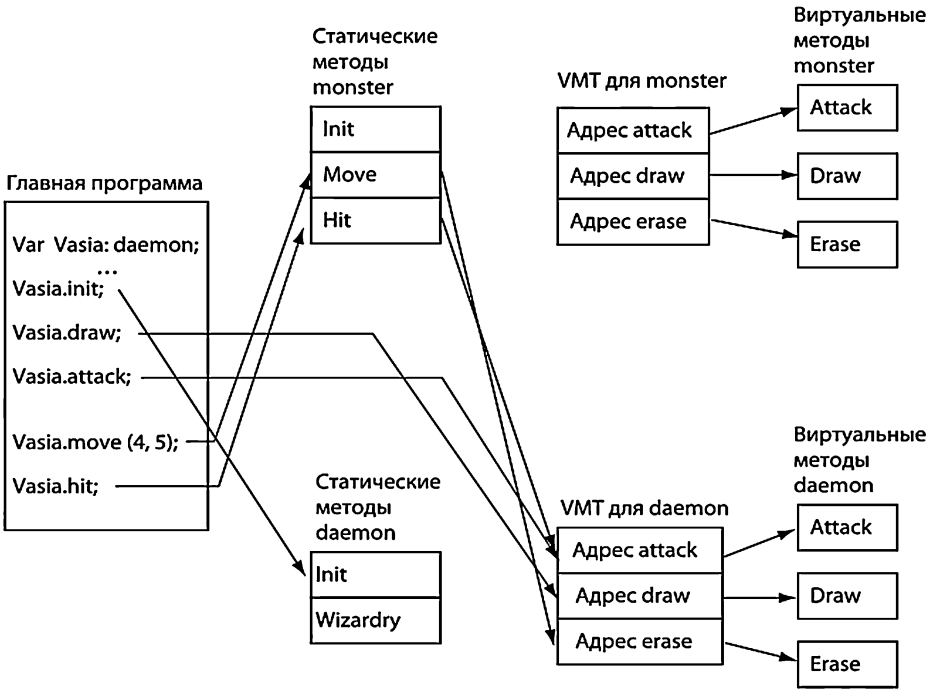

Рис. 6.7. **Механизм виртуальных методов**

Правила описания виртуальных методов.

1. Если в объекте метод определен как виртуальный, во всех потомках он также должен быть виртуальным.
2. Заголовки всех одноименных виртуальных методов должны совпадать.
3. Переопределять виртуальный метод в каждом из потомков не обязательно: если он выполняет устраивающие потомка действия, он будет унаследован.

4. Объект, имеющий хотя бы один виртуальный метод, должен содержать конструктор.

5. При описании объектов рекомендуется определять как виртуальные те методы, которые в его потомках будут реализовываться по-другому.

Объекты в динамической памяти

Для хранения объектов в программах чаще всего используется динамическая память, поскольку это более эффективно. Благодаря расширенной совместимости типов можно описать указатель на базовый класс и хранить в нем ссылку на любой его объект-потомок, что в сочетании с виртуальными методами позволяет единообразно работать с различными классами иерархии. Из объектов и указателей на них создают различные динамические структуры.

Для выделения памяти под объекты используются процедура и функция new. Например, если определены указатели:

```
type
  pmonster = ^monster;
  pdaemon = ^daemon;
var pm : pmonster;
  pd : pdaemon;
```

можно создать объекты с помощью вызовов:

```
new (pm); {или pm:= new (pmonster);}
new (pd); {или pd:= new (pdaemon);}
```

Так как после выделения памяти объект обычно инициализируют, для удобства определены расширенные формы new со вторым параметром — конструктором объекта: new (pm, init (1, 1, 1, 1);

```
new(pm, init(1, 1, 1, 1);
{или pm := new(pmonster, init(1, 1, 1, 1));}
new(pd, init(1, 1, 1, 1, 1);
{или pd := new(pdaemon, init(1, 1, 1, 1, 1));}
```

Обращение к методам динамического объекта выполняется по обычным правилам:

```
pm^.draw; pm^.attack;
```

С объектами в динамической памяти часто работают через указатели на базовый класс, т. е. описывают указатель базового класса, а инициализируют его, создав объект производного класса:

```
pm : = new
(pdaemon, init (1, 1, 1, 1, 1));
```

Такие объекты называют *полиморфными*. Они используются, для того чтобы можно было работать с объектами разных классов единообразно.

Деструкторы

Для освобождения памяти, занятой объектом, применяется процедура Dispose:

```
Dispose (pm);
```

Она освобождает количество байтов, равное размеру объекта, соответствующего типу указателя. Если в указателе хранится ссылка на объект производного класса, будет освобождено неверное количество байтов. Для корректного освобождения памяти из-под полиморфных объектов следует использовать вторым параметром Dispose специальный метод — *деструктор*. В документации по Borland Pascal ему рекомендуется давать имя done:

```
destructor monster.done;
begin
end;
...
Dispose (pm, done);
```

В деструкторе можно описывать любые действия, необходимые для конкретного объекта, например закрытие файлов. Деструктор может быть пустым. Для объектов, содержащих динамические поля, в нем записываются операторы освобождения памяти для этих полей.

Компилятор по служебному слову destructor вставляет в конец тела метода операторы получения размера объекта из VMT. Деструктор передает этот размер процедуре Dispose, и она освобождает количество памяти, соответствующее фактическому типу объекта. Деструкторы обязательно использовать только для динамических полиморфных объектов. В объекте можно определить несколько деструкторов (в этом случае они должны иметь разные имена).

Литература

1. *Вирт, Н.* Алгоритмы и структуры данных. Новая версия для Оберона / Н. Вирт. — Москва : ДМК Пресс, 2010. — 274 с.

2. *Голицына, О. Л.* Основы алгоритмизации и программирования / О. Л. Голицына, И. И. Попов. — Москва : Форум, 2008. — 432 с.

3. *Макарова, Н. В.* Информатика : учебник для вузов / Н. В. Макарова. — Санкт-Петербург : Питер, 2013. — 576 с.

4. *Макконнелл, С.* Совершенный код / С. Макконнелл. — Санкт-Петербург : Питер, 2009. — 896 с.

5. *Орлов, С. А.* Технологии разработки программного обеспечения / С. А. Орлов, Б. Я. Цилькер. — Санкт-Петербург : Питер, 2012. — 608 с.

6. *Павловская, Т. А.* Паскаль. Программирование на языке высокого уровня : учебник для вузов / Т. А. Павловская. — 2-е изд. — Санкт-Петербург : Питер, 2010. — 464 с.

Наши книги можно приобрести:

Учебным заведениям и библиотекам:
в отделе по работе с вузами
тел.: (495) 744-00-12, e-mail: vuz@urait.ru

Частным лицам:
список магазинов смотрите на сайте urait.ru
в разделе «Частным лицам»

Магазинам и корпоративным клиентам:
в отделе продаж
тел.: (495) 744-00-12, e-mail: sales@urait.ru

Отзывы об издании присылайте в редакцию
e-mail: gred@urait.ru

Новые издания и дополнительные материалы доступны на образовательной платформе «Юрайт» urait.ru, а также в мобильном приложении «Юрайт.Библиотека»

Учебное издание

Трофимов Валерий Владимирович,
Павловская Татьяна Александровна

АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

Учебник для вузов

Формат $70\times100\,^{1}/_{16}$.
Гарнитура «Charter». Печать цифровая.
Усл. печ. л. 8,38.

ООО «Издательство Юрайт»
111123, г. Москва, ул. Плеханова, д. 4а.
Тел.: (495) 744-00-12. E-mail: izdat@urait.ru, www.urait.ru