Саратовский государственный университет им. Н. Г. Чернышевского

Т. В. Диканев, С. Б. Вениг, И. В. Сысоев

ПРИНЦИПЫ И АЛГОРИТМЫ ПРИКЛАДНОГО ПРОГРАММИРОВАНИЯ

Учебное пособие для студентов, обучающихся на факультете нано- и биомедицинских технологий

Саратов
Издательство Саратовского университета
2012

УДК 519.683, 372.862
ББК 32.973-018.2я73
Д45

Диканев, Т. В.

Д45 Принципы и алгоритмы прикладного программирования : учебное пособие для студентов, обучающихся на факультете нано- и биомедицинских технологий / Т. В. Диканев, С. Б. Вениг, И. В. Сысоев. – Саратов : Изд-во Сарат. ун-та, 2012. – 140 с. : ил.

ISBN 978-5-292-04146-7

Идея книги – объяснить основные принципы и базовые алгоритмы написания прикладных программ людям, для которых программирование не является профессией, а только средством повысить продуктивность своего труда. При этом упор делается не на знание конкретного языка, а на формирование алгоритмического мышления. Овладение программированием невозможно без практики, для чего в книге имеется обширный набор заданий.

Для студентов естественно-научных направлений подготовки.

Рекомендуют к печати:

кафедра динамического моделирования и биомедицинской инженерии
факультета нано- и биомедицинских технологий
Саратовского государственного университета
доктор физико-математических наук, профессор *В. В. Астахов*

Работа издана по тематическому плану 2012 года
(утвержден на Ученом совете Саратовского государственного университета, протокол № 2 от 31 января 2012 г.)

УДК 519.683, 372.862
ББК 32.973-018.2я73

ISBN 978-5-292-04146-7

Введение

Данное пособие представляет собой начальный курс программирования для студентов 1-го курса кафедры динамического моделирования и биомедицинской инженерии факультета нано- и биомедицинских технологий СГУ. Основным его отличием от большинства других книг для начинающих является упор не на язык программирования (изложением которого обычно и ограничиваются), а на выделение различных алгоритмических приемов.

Возможно, наиболее важной частью курса является набор задач, включающий как простые примеры для делающих первые шаги в программировании, так и более сложные, приближающиеся к олимпиадному уровню. При формировании набора задач учитывался 6-летний опыт преподавания программирования на нашей кафедре, в ходе которого типичные ошибки в мышлении студентов анализировались и для их лучшего исправления придумывались дополнительные задачи.

В качестве языка программирования используется старый добрый Паскаль. На хорошо знакомую критику, что данный язык устарел и следует изучать популярные Java, C++ и т.п., отвечаем: главной целью курса является выработка навыков алгоритмического мышления и хорошего стиля при процедурном программировании. Специфические именно для языка Паскаль вещи занимают в данном пособии совершенно незначительное место. Упомянутые навыки мы считаем необходимой базой, без которой невозможен переход к изучению объектно-ориентированных языков и других современных технологий программирования.

Выбор именно Паскаля обусловлен, во-первых, простотой начального освоения – у людей, впервые столкнувшихся с программированием, хватает проблем помимо разбирательств с дебрями синтаксиса; во-вторых, многие стилистически правильные вещи возведены в Паскале в ранг обязательных правил.

Заметим, что между знанием языка и умением программировать разница примерно такая же, как между знанием слов и умением красиво и убедительно говорить. Язык Паскаль в основе содержит не более сотни слов и выучить их за сравнительно короткое время способен любой. Однако как только за изложением даже простейших языковых конструкций следует предложение применить их на практике, у большинства новичков начинается ступор и возникает непонимание, что же делать?

Хотим предупредить, что данное состояние совершенно естественно. Не следует думать, что дело в вашей неспособности или в недостатках данного пособия (мол, вам чего-то не рассказали). Будьте уверены, с вами все нормально и все, что надо, вы знаете. Дело в том, что искусство составлять решение задачи из того небольшого набора доступных в языке команд требует особого мыслительного навыка – алгоритмического мышления, которым большинство людей изначально не обладают. Причем это именно навык, а не какой-то набор знаний, который можно передать словами. Единственный путь к его обретению – решение задач.

Таким образом, начальный ступор и непонимание есть неизбежная ступень, которую вам придется преодолеть. Скорее всего, кроме личного упорства вам здесь ничто не поможет. Бесполезно просить товарищей или преподавателя решить вам задачу. Знание, как именно она решается, не даст вам решительно ничего. Важно не оно, а путь его получения, который будет скрыт от вас.

Подумайте, как учат ходить маленьких детей? Очевидно, им не рассказывают про биомеханику и не говорят, какие группы мышц следует напрягать. Вы – такие же младенцы в программировании и первые шаги вам предстоит сделать самостоятельно.

Выбор среды программирования

Для написания программ на языке Паскаль можно использовать несколько различных компиляторов и сред программирования. Они отличаются как функционалом, так и удобством. При выполнении заданий из данного учебного пособия мы рекомендуем использовать среду Geany (доступна по адресу http://www.geany.org/Download/Releases) и компилятор FreePascal (http://freepascal.org/download.var). Можно также воспользоваться средой PascalABC.NET со встроенным компилятором (скачать можно по адресу http://pascalabc.net/ssyilki-dlya-skachivaniya.html).

1. FreePascal – это активно развиваемый и достаточно стабильный компилятор Паскаля с открытым исходным кодом. Он полностью доступен для бесплатной загрузки, в том числе с исходными кодами. Основные преимущества:

- отсутствие существенных ошибок. Это важное качество при обучении – можно быть уверенным, что когда что-то не работает, ошибка именно ваша, а не разработчиков сред;
- поддержка многих современных конструкций (в том числе практически полная поддержка конструкций Delphi 7 и некоторых более поздних версий) и богатая стандартная библиотека, позволяющая переходить к более сложным программам, не меняя средств разработки;
- кроссплатформенность: доступность на Windows, Linux, Mac OS X, FreeBSD, что может быть важно для образовательных учреждений в свете постепенного их перехода на использовангие СПО;

– скорость и низкое потребление памяти: FreePascal – современный быстрый язык, что не так важно при обучении программированию, но может быть существенно для тех, кто в дальнейшем будет изучать численные методы и писать уже настоящие, а не учебные программы. Кроме того, это позволяет использовать FreePascal на слабых и старых машинах, которых немало в образовательных учреждениях.

Основные недостатки:

– стандартная среда разработки малопригодна для использования. Именно поэтому мы рекомендуем пользоваться Geany;
– на платформе Windows есть некоторые ограничения на использование русских букв в именах файлов и путях к ним, а также при выводе в терминал;
– вывод сообщений об ошибках дается на английском языке. Для профессионалов это норма, но для начинающих может стать неприятным сюрпризом.

2. PascalABC.NET – это современная бесплатная среда, разработанная на факультете математики, механики и компьютерных наук Южного федерального университета специально для целей обучения. К ее достоинствам можно отнести:

– простой интерфейс (ничего, что не потребуется при начальном обучении);
– русскоязычные сообщения об ошибках, подсказки и справочная система. Хотя и говорят, что программист обязан знать английский язык, однако давайте смотреть правде в глаза: когда все по-русски – заметно удобнее;
– неплохая выверенность. За 2 года использования на кафедре ошибки встречались нечасто. Большинство из них исправлено в новых версиях. Совместимость с современным языком Delphi.

Недостатки также имеют место:

– не всегда хорошо документированные мелкие отличия от наиболее распространённого диалекта Pascal - Delphi в языковых конструкциях;
– низкая скорость выполнения и существенные требования к объёму памяти. Это незаметно на более-менее современных компьютерах в учебных задачах, но на старых или слабых машинах, таких как нетбуки и неттопы, может стать проблемой;
– отсутствие компиляции – вы не сможете сделать выполняемый файл (часто называемый также exe-шник или бинарник) и перенести на другой компьютер, для исполнения программ необходимо установить среду разработки;
– отсутствие поддержки в Linux и Mac OS X, что иногда может стать препятствием в учебных учреждениях и не обрадует поклонников MacBook’ов, iMac’ов и заядлых линуксоидов.

Перечислим для полноты картины и другие среды, которые также можно использовать.

3. Borland Pascal 7.0

Это ставший классическим компилятор и среда программирования, часто используемая при обучении в школах и университетах до сих пор (упрощённая и более дешёвая версия с урезанной стандартной библиотекой называлась Turbo Pascal). К достоинствам можно отнести выверенность – если что-то не работает, можно быть на 99,99% уверенным, что виноват программист, а не «глюки» среды.

Основной недостаток – устаревший интерфейс (последняя версия выпущена в 1993 году). Привыкшие к традиционным для Windows интерфейсам студенты поначалу испытывают определенные трудности с самыми простыми операциями (сохранение/загрузка, копирование/вставка, навигация между закладками и т.п.). Кроме того, это 16-битный компилятор, который на современных версиях Windows просто не работает. У Borlan Pascal есть проблема легального использования: этот компилятор уже давно не продаётся, но свободное его копирование по-прежнему запрещено законами об авторском праве, т.е. стать его обладателем легально фактически нельзя.

4. Delphi

Это современная профессиональная среда разработки. Для выполнения заданий в ней требуется создавать так называемые консольные приложения. Недостатком можно счесть сложный интерфейс, пугающий начинающего обилием кнопочек, окошек и закладок. При этом большинство предоставляемых возможностей на начальном этапе обучения программированию все равно не понадобятся. Кроме того, современные версии Delphi требовательны к ресурсам и стоят недёшево даже для образовательных целей.

5. Lazarus

Lazarus – бибилиотека компонентов для создания визульных программ и среда разработки для FreePascal. Lazarus + FreePascal иногда рассматривают как бесплатный аналог Delphi, хотя такой подход отражает только один из аспектов их использования. Применение Lazarus на начальном этапе нецелесообразно по тем же причинам, что и Delphi. Кроме того, он не свободен от ошибок и ограничений, в результате чего иногда не сразу можно понять, является ли сбой в работе программы следствием ошибки программиста или «глюком» среды (надо также отметить, что Lazarus в целом лучше работает в Linux, чем в Windows, особенно это касается ситуации, когда права пользователя ограничены).

Дальнейшее изложение будет вестись таким образом, чтобы задания могли быть выполнены в любой из сред: Geany+FreePascal или PascalABC.NET; при наличии отличий от Borland Pascal 7.0 или различий между этими средами будут даваться соответствующие пояснения.

1. Линейные программы: арифметические операторы, стандартные функции и ввод/вывод в текстовом режиме

1.1. Алгоритмы

Алгоритм – система четких однозначных указаний, которые определяют последовательность действий над некоторыми объектами и после конечного числа шагов приводят к желаемому результату. Запись алгоритма на языке программирования называется программой.

Чтобы расшифровать упомянутые в определении «некоторые объекты», рассмотрим простую схему работы компьютера (рис. 1.1).

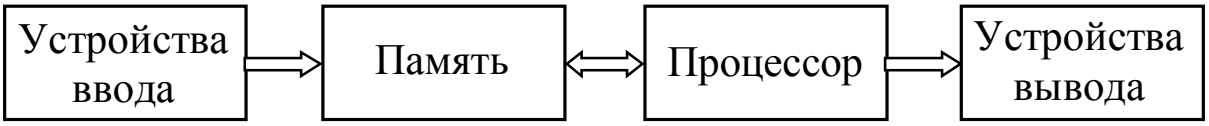


Рис. 1.1

Память хранит как данные (например, числа в двоичном формате), так и команды для процессора. Процессор, считывая из памяти команды, выполняет действия, которые заключаются в изменении содержимого ячеек памяти и командах, отдаваемых периферийным устройствам (например, устройствам вывода). Память можно представить себе как длинную ленту, разделенную на ячейки, в которых записаны либо нули, либо единицы (рис. 1.2).

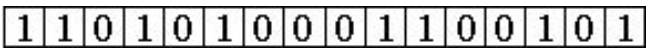


Рис. 1.2

«Некоторыми объектами» у нас будут ячейки памяти, содержимое которых мы будем менять, а также экран монитора, на который мы будем выводить информацию.

1.2. Переменные и их типы

Обратиться к содержимому ячеек памяти в Паскале можно, вводя *переменные*. Переменная – это область памяти, которой назначено имя (*идентификатор*).

Правила составления идентификаторов:

1) идентификатором может быть любое сочетание латинских букв, цифр и знака подчеркивание;

2) цифра не может быть первым символом;

3) большие и малые латинские буквы не различаются (идентификатор A1 эквивалентен идентификатору a1).

Примеры правильных идентификаторов: A, x1, x_1, _b1, SxCf.

Примеры неправильных: 1b, a–b.

Информация, записанная в ячейках памяти с помощью двоичного кода, может быть самого разного рода: двоичное представление чисел, коды текстовых символов, строк и т.д. Какие операции над ячейками памяти допустимы, зависит от типа хранимой в ней информации. Например, хранимое в памяти число можно возвести в квадрат. Но если в тех же ячейках памяти хранится закодированный текст (скажем, ваше имя), то что есть квадрат от вашего имени – неясно. Формальное выполнение тех же действий, что и при возведении числа в квадрат, приведет к бессмысленному результату. В силу этого в языках высокого уровня существует строгая типизация переменных. Иными словами, прежде чем использовать переменную, необходимо указать ее тип. Набор допустимых операций зависит от выбранного вами типа.

Перечислим основные типы, используемые в Паскале:

Integer – целый тип. Его переменные могут хранить целые числа в диапазоне -2147483648 до 2147483647 (это -2^{31} и $2^{31}-1$).

Real – вещественный тип. Так называемые числа с плавающей точкой. Может быть обычной десятичной дробью (например, 1234.543), а также содержать порядок – символ «e» и какое-либо число за ним, например, 1.2345e3. Такая запись означает, что число 1.2345 нужно умножить на 10^3. Максимальное количество цифр в числе 15, порядок может быть в диапазоне от -308 до 308.

Char – символьный тип. Значением этой переменной может быть одиночный символ – буква латинского алфавита (большие и малые буквы здесь различаются), цифра или какой-либо из специальных символов.

String – строка. Значения переменных – наборы символов.

Boolean – логический тип. Переменная может принимать два значения: *true* (истина) и *false* (ложь). Такие значения могут быть, например, у логических выражений наподобие «x>2». Если истинно, что x>2, то выражение принимает значение *true* иначе значение *false*.

Чтобы указать тип переменной в Паскале, нужно написать ключевое слово **var**, затем имя переменной, двоеточие и тип. Например:

```
var
x: integer;
y, z: real;
a22: char;
b_b: string;
```

Задание типа в Паскале называется описанием переменной. Отдельные описания разделяются символом точка с запятой. Если необходимо несколько переменных одного типа, их можно писать через запятую.

Особенности среды Borland Pascal

Если в качестве среды разработки вы используете Borland Pascal, то следует иметь в виду следующие особенности:

- допустимые значения типа integer будут лежать в диапазоне от −32768 до 32767 (это −215 и 215−1);
- значения типа real могут содержать не более 11 цифр, а допустимые порядки варьируются от −38 до 38;
- строка (переменная типа string) может содержать не более 255 символов.

1.3. Операторы

Всякая программа представляет собой последовательность команд или инструкций. Элементарные команды, из которых конструируются более сложные, называются *операторами*.

Одно из самых фундаментальных действий, которые можно сделать с переменной, – это присваивание значения. Соответствующая инструкция имеет вид

```
x := 2;
```

Символ «:=» (двоеточие и равно) называется оператором присваивания. Слева от оператора должна стоять переменная, справа – выражение, значение которого имеет тот же тип, что и переменная.

Примеры неправильного использования оператора присваивания:

```
x := 2.5;     {переменной x целого типа присваивается
              нецелое значение}
x := y;       {переменной целого типа присваивается
              значение переменной вещественного типа}
```

Однако инструкция `y := x;` допустима, так как целые числа являются подмножеством вещественных. Чтобы присвоить значения переменным символьного и строкового типа, соответствующий символ или строку надо взять в одинарные кавычки:

```
a22 := 'x';
b_b := 'Hello, world!';
```

Отдельные инструкции в Паскале (а каждое присваивание является отдельной инструкцией) разделяются символом «;».

Арифметические операторы: +, – , *, /, **div**, **mod**. Первые четыре обычные операции – сложение, вычитание, умножение, деление. **div** – взятие целой части от деления двух целых чисел, **mod** – взятие остатка от де-

ления двух целых чисел. Результат работы этих операторов может быть присвоен переменной:

```
x := 2*2;
y := (2+x)/5;
```

Следует помнить, что оператор деления «/» в Паскале всегда дает результат в виде вещественного числа, который не может быть присвоен переменной целого типа. Например, недопустима инструкция:

```
x := 4/2;
```

Вместо этого следует писать:

```
x := 4 div 2;
```

Сама переменная, которой присваивается значение, может использоваться в выражении, стоящем справа от оператора присваивания. Допустимы, например, следующие инструкции:

```
x := x+1; {увеличивает значение переменной x на 1}
x := 2*x*x;
```

и т.п.

1.4. Стандартные функции

Кроме арифметических операторов, в выражениях могут участвовать функции. У функций есть аргументы, и говорят, что функция возвращает значение. Аргументы пишутся в скобках вслед за именем функции, например, sin(y) – возвращает синус от значения переменной y. Возвращаемые значения можно присваивать переменным или использовать в выражениях:

```
z := sin(y);
y := (1+2*sin(y))/2;
```

Для каждой функции необходимо знать допустимый тип ее аргументов. Например, аргументом синуса должно быть выражение либо целого, либо вещественного типа. Невозможно взять синус от строки или символа. В то же время, например, функция length(s), определяющая длину строки, может быть взята только от строковой переменной.

Кроме того, возвращаемое функцией значение тоже имеет определенный тип. Функция sin возвращает вещественное значение и его нельзя присвоить целочисленной переменной, а скажем, функция length(s) возвращает целочисленное значение.

Перечислим несколько наиболее распространенных стандартных функций:

round(y) – округление числа. Аргумент целое или вещественное число. Возвращаемое значение целого типа;

trunc(y) – отбрасывание дробной части. Возвращаемое значение целого типа;

sin(y), cos(y) – синус и косинус;

ln(y) – натуральный логарифм;

exp(y) – экспонента;

sqr(y) – возведение в квадрат. Тип возвращаемого значения зависит от типа аргумента. Если аргумент был целым, то результат будет целым. Иначе результат будет вещественным;

sqrt(y) – квадратный корень. Возвращаемое значение вещественного типа;

abs(y) – модуль. Тип возвращаемого значения зависит от типа аргумента. Если аргумент был целым, то результат будет целым. Иначе результат будет вещественным;

arctan(y) – арктангенс.

У функций может быть более одного аргумента. С примерами таких функций мы познакомимся позднее. Есть функции вообще без аргументов, например:

random – возвращает случайное число в диапазоне от 0 до 1;

pi – возвращает число π.

Аргументом функции может быть не только одна переменная, но и произвольное выражение, в частности, содержащее другие функции. Например:

```
x := 2+round(sin(2*pi*y)+2);
```

1.5. Структура программы

Структура программы на Паскале представлена ниже:

```
program <Имя программы>;
<Раздел описаний>
begin
<Раздел операторов>
end.
```

Программа на Паскале начинается с ключевого слова **program**, после которого идет имя программы (любой правильный идентификатор) и ставится точка с запятой. Затем идет раздел описаний. Здесь, в частности, располагается раздел описания переменных. Все инструкции располагаются в разделе операторов, в том числе операторы присваивания. Пример программы:

```
program MyFirstProgram;
var
   x: integer;
begin
   x := 2;
end.
```

1.6. Ввод/вывод в текстовом режиме

Приведенная в параграфе 1.5 программа запишет в некоторую область памяти число 2, однако внешне это никак не проявится. Нормальная программа должна уметь получать информацию от пользователя и сообщать ему результаты своей работы.

В простейшем случае вывод информации осуществляется процедурами write и writeln. В обоих случаях выводятся текст или значения переменных. Отличие этих двух процедур в том, что последовательные вызовы write будут печатать информацию в одной строке, а writeln после каждого вызова переводит курсор на следующую строку. Примеры вызова процедур вывода:

```
writeln('Hello');
```

Печатается слово 'Hello'.

```
writeln(x);
```

Напечатается значение переменной x.

```
x := 2;
writeln('x = ', x);
```

В одну строчку напечатаются строка «x = » и значение переменной x, т. е. в результате будет напечатано «x = 2».

```
x := 2;
y := 3;
writeln(x, y);
```

В одну строчку напечатаются значения переменных x и y, т. е. «23».

```
writeln(x, '   ', y);
```

Между значениями x и y будет располагаться пробел.

```
writeln(x);
writeln(y);
```

Значения x и y будут напечатаны на разных строках.

```
writeln(2*x+y);
```

Будет напечатано значение выражения 2*x+y.

```
writeln;
```

Вызов writeln без параметров приводит к переходу на новую строку. Несколько таких вызовов подряд вставят в выводимый текст несколько пустых строк.

Общее правило таково: write и writeln могут печатать значения переменных и произвольные текстовые строки. При этом текстовые строки должны быть взяты в кавычки. Одной процедурой можно вывести не-

сколько значений переменных и строк текста, перечислив выводимые элементы через запятую.

Ввод информации осуществляется с помощью процедуры

`readln(<список переменных>);`

которая позволяет ввести с клавиатуры значения нескольких переменных. Например:

```
readln(x);
```

Выполнение программы приостановится, пока пользователь не введет значения переменной x и не нажмет Enter.

```
readln(x, y, z);
```

Программа ожидает ввода значений трех переменных. После каждого ввода следует жать Enter.

```
write('x = ');
readln(x);
```

Курсор будет мигать не на пустой строке, а на строке, содержащей приглашение вида «x = ».

```
readln;
```

В отсутствие параметров процедура просто приостанавливает выполнение программы до нажатия пользователем клавиши Enter.

Используем наши новые знания, написав программу, запрашивающую у пользователя два числа и печатающую их сумму:

```
program Summa;
var
   x, y: integer;
begin
   write('x = ');     {вывод текста 'x = '}
   readln(x); {ввод значения переменной x пользовате-
               лем программы}
   write('y = ');     {вывод текста 'y = '}
   readln(y); {ввод значения переменной y пользовате-
               лем программы}
   writeln('Summa = ', x+y);  {печать результата}
end.
```

1.7. Задачи на составление арифметических выражений

Хотя для наглядности речь в задачах может идти о столбах, шагах или распиливании бревна, полностью аналогичные задачи программистам приходится решать на каждом шагу (при работе с массивами, при вычислении количества шагов в циклах и т.д.) Все выражения, которые вам требуется составить, взяты из реальной программистской практики.

Прорешать данные задачи также полезно в связи с тем, что многие не умеют их решать в общем виде, когда вместо конкретных чисел используются буквенные обозначения. Кроме того, большинство выпускников обычных школ плохо представляют себе, что значит думать над задачей. Они могут прилежно заучить правила написания и смысл операторов языка, но когда надо изобрести способ решить задачу с их помощью, не понимают, что же им делать.

В данном случае мы имеем дело с простейшей ситуацией – по сути, требуется придумать программу, состоящую из одной строки. Набор операций дан, требуется их скомбинировать так, чтобы получился необходимый результат. Решение данных задач позволит выработать правильный подход и настрой для работы в ситуации, когда «неизвестно, что делать».

Задачи

Используя арифметические операторы (+, -, *, /, **div**, **mod**), а также функции round(), trunc() и abs(), составьте арифметические выражения для вычисления следующих величин:

1. *n*-е четное число (первым считается 2, вторым 4 и т.д.).
2. *n*-е нечетное число (первое – 1, второе – 3 и т.д.).
3. В очереди стоит *n* людей, сколько человек находится между *i*-м и *k*-м в очереди?
4. Сколько нечетных чисел на отрезке (*a*, *b*), если *a* и *b* – четные? *a* и *b* – нечетные? *a* – четное, *b* – нечетное?
5. Сколько полных минут и часов содержится в *x* секундах?
6. В доме 9 этажей, на каждом этаже одного подъезда по 4 квартиры. В каком подъезде и на каком этаже находится *n*-я квартира?
7. Старинными русскими денежными единицами являются: 1 рубль – 100 копеек, 1 гривна – 10 копеек, 1 алтын – 3 копейки, 1 полушка – 0,25 копейки. Имеется *A* копеек. Запишите выражения для представления имеющейся суммы в рублях, гривнах, алтынах и полушках.
8. Стрелка прибора вращается с постоянной скоростью, совершая *w* оборотов в секунду (не обязательно стрелка прибора, это может быть волчок в игре «Что? Где? Когда?» и т.п.). Угол поворота стрелки в нулевой момент времени примем за 0. Каков будет угол поворота через *t* секунд?
9. Вы стоите на краю дороги и от вас до ближайшего фонарного столба *x* метров. Расстояние между столбами *y* метров. На каком расстоянии от вас находится *n*-й столб?
10. Та же ситуация, что и в предыдущей задаче. Длина вашего шага *z* метров. Мимо скольких столбов вы пройдете, сделав *n* шагов?
11. *x* – вещественное число. Запишите выражение, позволяющее выделить его дробную часть.
12. *x* – вещественное число. Запишите выражение, которое округлит его до сотых долей (останется только два знака после запятой).

13. n – целое число. Запишите выражение, позволяющее узнать его последнюю цифру.
14. n – четырехзначное целое число. Запишите выражение, позволяющее узнать его первую цифру.
15. Оператор **div** в Паскале работает только для целых чисел. Составьте выражение, позволяющее получить целую часть от деления вещественных чисел.
16. Выразите операцию **mod** через другие арифметические операции.
17. x – вещественное число. Запишите выражение, которое даст +1, если $x>0$ и –1, если $x<0$ (при $x = 0$ выражение будет не определено).
18. n и m – целые числа. Запишите выражение, которое давало бы 0, если n кратно m, и 1, если не кратно.
19. От бревна длиной L отпиливают куски длиной x. Сколько кусков максимально удастся отпилить?
20. Бревно длиной L распилили в n местах. Какова средняя длина получившихся кусков?
21. Резиновое кольцо диаметром d разрезали в n местах. Какова средняя длина получившихся кусков?
22. На прямой через равные промежутки располагается n точек. Расстояние от первой до последней равно L. Чему равно расстояние от первой точки до i-й? от k-й до последней? от i-й до k-й?
23. Известно, что приближенные формулы для вычисления синуса и косинуса работают тем точнее, чем меньше значение аргумента. Поскольку синус и косинус 2π-периодические функции ($\sin(x) = \sin(x+2\pi n)$, где n – любое целое число), то можно вычисление синуса от любого аргумента привести к вычислению синуса от аргумента, лежащего в диапазоне от 0 до 2π. Запишите формулы, позволяющие:

(а) привести положительный угол x в диапазон от 0 до 2π;

(б) аналогично привести отрицательный угол в тот же диапазон.

24. Пусть дано трехзначное число x (например, 123). Составьте выражения, которые позволят вычислить первую, вторую и третью цифру этого числа (числа 1, 2 и 3 в примере). Для облегчения поиска решения имейте в виду, что для двузначного числа первая цифра дается выражением

```
d1 := x div 10;
```

а вторая выражением

```
d2 := x mod 10;
```

Контрольная работа 1

1. Дайте определение алгоритма.
2. Каким ключевым словом открывается раздел описания переменных?
3. Объявите переменную целого типа.

4. Какие присваивания в приведенной программе являются недопустимыми (укажите соответствующие номера строк)?

```
1     var
2        x: integer;
3        y, z: real;
4        a22: char;
5     begin
6        x := y;
7        y := x;
8        y := (x*z+5)*ln(x);
9        y := x*y;
10       x := x*y;
11       x := x/2;
12       x := x+5;
13       x := x*x*x*x*x;
14       a22 := round(x);
15       x := round(x);
16       a22 := 'x';
17       y := y div 2;
18    end.
```

5. Вычислите значения следующих выражений или укажите, что вычисление невозможно:

а) `25 div 6`
б) `-25 mod 6`
в) `25.1 mod 5`
г) `24 mod 0`
д) `3 mod 5`
е) `3 div 5`
ж) `14 mod 1`
з) `sqrt(ln(1)-1)`
и) `-4 / 2*2`
к) `3/2 div 5`
л) `3 div 5/2`
м) `trunc(-14)`
н) `round(-5.5)`
о) `trunc(14.234e2)`

6. Чему равны переменные после выполнения следующих фрагментов программ:

а)
```
x:=3.14159;
x:=round(100*x)/100;
```
б)
```
x:=11;
x:=trunc(x/2);
```
в)
```
x:=22;
x:=(x-x)*x;
```
г)
```
x:=193745;
y:=x+1;
x:=(x+y) mod 2;
```
д)
```
x:=2;
y:=6;
x:=y;
y:=x;
```
е)
```
a:=1;
b:=2;
c:=a;
a:=b;
b:=c;
```
ж)
```
a:=11;
b:=45;
a:=b-a;
b:=a-b;
a:=a-b;
```

7. Нарисуйте графики функций:

а) `y = round(sin(x))`
б) `y = trunc(2*sin(x))`
в) `y = abs(cos(x))`
г) `y = trunc(sin(x)+1)`

Задание 1. Линейные программы, арифметические операторы

1. Создайте программу, печатающую при запуске текст «Hello, World!» (традиционный текст первой программы при изучении языка программирования, ваша первая программа приветствует мир). Опробуйте на ней возможности среды разработки. Запустите ее (F9), просмотрите результат выполнения, сохраните на диск (Ctrl – S), загрузите с диска снова (Ctrl – O). Пользуясь копированием через буфер (Ctrl – C, Ctrl – V), сделайте так, чтобы программа выводила слово «Hello» 20 раз.

2. Напишите программу, запрашивающую у пользователя два числа и печатающую их сумму. Убедившись, что программа работает, намеренно допустите ошибку, не поставив точку с запятой после какого-нибудь оператора. Обратите внимание на сообщение об ошибке, выданное средой. Допустите другую ошибку, записав неправильно имя процедуры вывода (например, writln вместо writeln), снова прочитайте текст сообщения об ошибке. Попытайтесь воспользоваться переменной, предварительно ее не описав. Опишите переменную типа integer, попытайтесь присвоить ей нецелое значение.

3. Создайте программу, решающую квадратные уравнения. Программа должна запрашивать значения коэффициентов и печатать вычисленные корни.

4. Имеется девятиэтажный дом, на каждую лестничную площадку выходит 4 квартиры. Создайте программу, которая по номеру квартиры определяет номер подъезда и этаж.

5. Если дано трехзначное число, например 123, его можно представить в виде 3+2*10+1*10*10. Воспользовавшись этой информацией, создайте программу, которая, получая от пользователя трехзначное число, будет определять, из каких цифр оно состоит, и выводить их через пробел (например, 1_2_3).

6. Напишите программу, запрашивающую у пользователя два момента времени (количество часов, минут и секунд) и сообщающую число секунд, прошедшее между этими двумя моментами.

7. Напишите программу, запрашивающую у пользователя время и сообщающую угол поворота минутной и часовой стрелки в градусах и радианах (помните, что тригонометрические функции в Паскале работают с радианами).

Особенности среды Borland Pascal

Если в качестве среды разработки вы используете Borland Pascal, то следует иметь в виду следующие особенности:

- после запуска среды необходимо создать новый файл (меню File/New);
- сохранение и загрузка файла производятся с помощью клавиш F2 и F3;

– копирование и вставка производятся сочетаниями клавиш Ctrl – Ins и Shift – Ins (вместо привычных Ctrl – C, Ctrl – V);
– после того как программа чего-то напечатает, она завершится, и вы не увидите никакого результата, пока не нажмете Alt – F5. Чтобы не делать этого каждый раз, рекомендуется помещать в конец программы вызов процедуры readln:

```
    ...
    readln;
end.
```

Это не позволит программе завершиться, пока вы не нажмете Enter, так что вы сразу сможете видеть результат ее работы;
– сообщения об ошибках вы будете получать на английском языке. Важно перевести, понять и запомнить их содержание.

2. Логические выражения и условный оператор

2.1. Переменная логического типа

Переменные логического (или булевского) типа могут принимать два значения – true (истина) или false (ложь). Описываются они следующим образом:

```
var
   b1, b2: boolean;
```

Допустимы присваивания вида:

```
b1 := true;
b2 := false;
b1 := b2;
```

и т.п.

2.2. Операторы сравнения

Имеется 6 операторов сравнения: >, <, =, <=, >=, <>. С их помощью записываются простые логические выражения, принимающие значения true или false. Эти значения можно присваивать логическим переменным.

Пусть, например, описаны переменные:

```
var
   b: boolean;
   x: real;
```

Допустимы следующие присваивания:

```
b := 2 > 5;
b := x > 2;
b := (sqrt(x)-1)/2 > sqr(x);
```

и т.п.

Первый из приведенных операторов запишет в переменную b значение false. Результат работы остальных зависит от значения переменной x.

Операторы сравнения применимы и к самим логическим переменным и выражениям. При этом считается, что true больше, чем false. Так, например, возможны выражения:

```
b := (2 > 5) < (6 > 2);   {в переменную b будет записано значение true}
b := b > false;           {значение переменной b не изменится}
b := b = true;{снова переменная b останется такой, какая была}
```

Иногда, когда требуется проверить равенство сразу трех величин, студенты ошибочно пишут условие вида

$$x = y = z.$$

Паскаль интерпретирует это выражение следующим образом. Первое равенство x = y даст значение true или false, и уже это значение будет сравниваться со значением переменной z. Если z не является переменной логического типа, то среда выдаст сообщение об ошибке. Действительно, нет смысла проверять равенство логического значения и, например, числового. Чтобы проверить равенство сразу трех чисел, необходимо использовать логические операторы.

2.3. Логические операторы

Из логических переменных и выражений можно строить более сложные (составные) логические выражения с помощью логических операторов: **not** (*отрицание, логическое НЕ*), **or** (*логическое ИЛИ*) и **and** (*логическое И*).

Выражение **not** A (где A – логическая переменная или выражение) истинно тогда, когда выражение A ложно, и ложно, когда A истинно.

Выражение A **and** B истинно, когда одновременно истинны выражения A и B. Если хотя бы одно из этих выражений (A или B) ложно, то A **and** B ложно.

Выражение A **or** B истинно, когда любое из выражений A или B истинно, и ложно, когда оба исходных выражения ложны.

Правила работы логических операторов можно также задать с помощью *таблиц истинности*, в которых указывается истинность составного выражения, в зависимости от значений исходных простых выражений (табл. 1). Составное логическое выражение может содержать сколько угодно логических операторов. При этом в первую очередь выполняются все операторы сравнения (<, >, =, <=, >=, <>), затем логические отрицания (**not**), затем логическое И (**and**) и в последнюю очередь логическое ИЛИ (**or**). Выражения могут содержать скобки, которые влияют на приоритетность выполнения операций.

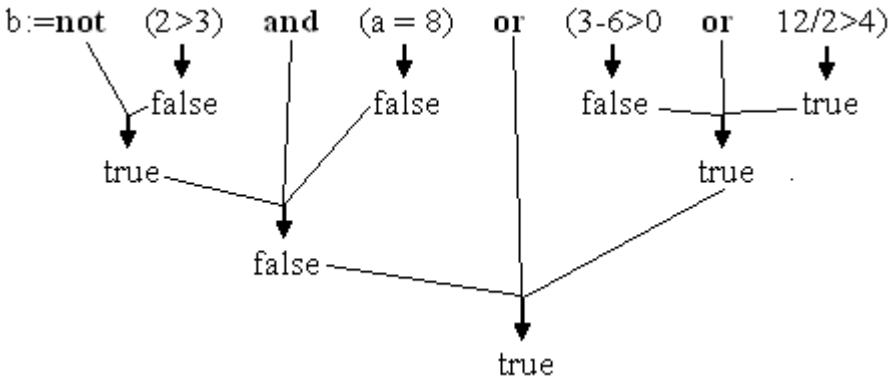
Таблица 1

Таблица логических операторов

A	not A	A	B	A and B	A or B
true	false	true	true	true	true
false	true	true	false	false	true
–	–	false	true	false	true
–	–	false	false	false	false

Пример вычисления сложного условия:

```
a:=5;
```



С помощью логических операторов мы, наконец, можем записать условие равенства сразу трех переменных. Правильный вариант имеет вид (x=y) **and** (y=z)

2.4. Задачи на составление логических выражений

Попробуйте самостоятельно составить логические выражения, принимающие значение true в перечисленных ниже случаях.

1) переменная x попадает в диапазон от –2 до 1 ($x \in [-2;1]$). Ниже данный диапазон показан на числовой оси (рис. 2.1):

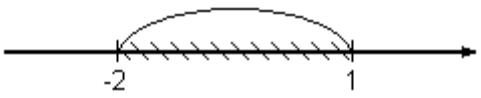


Рис. 2.1

2) переменная x лежит за пределами заданного диапазона, как показано на числовой оси (рис. 2.2):

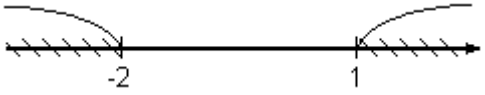


Рис. 2.2

3) переменная x лежит в одной из показанных на числовой оси областей (рис. 2.3):

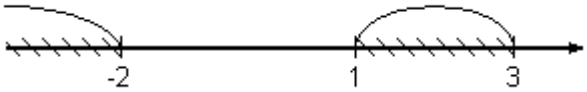
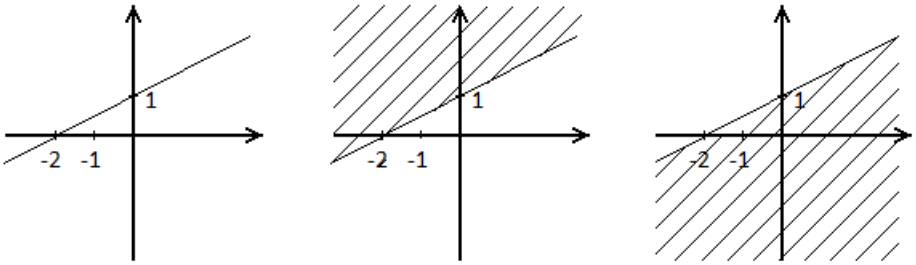


Рис. 2.3

4) запишите условия, истинные, когда точка с координатами (x, y) лежит точно на прямой, показанной на рис. 2.4, *а*, выше этой прямой (рис. 2.4, *б*) и ниже этой прямой (рис. 2.4, *в*):



а *б* *в*

Рис. 2.4

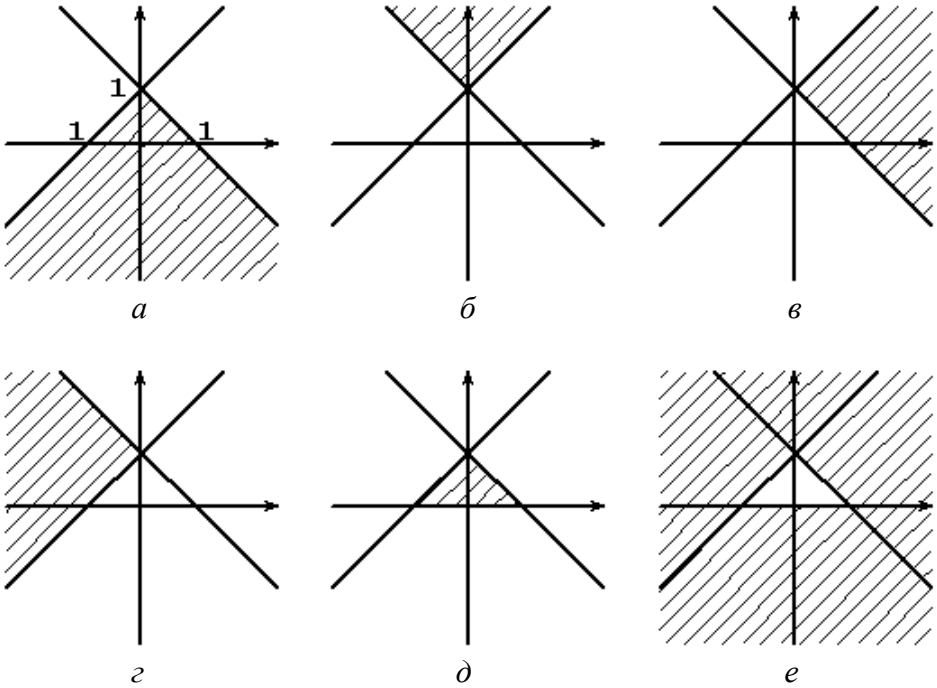
Указание. Чтобы записать уравнение прямой, проходящей через точки (x_1, y_1) и (x_2, y_2), в виде $y = kx + b$, необходимо решить систему уравнений:

$$\begin{cases} y_1 = kx_1 + b, \\ y_2 = kx_2 + b. \end{cases}$$

В результате приходим к уравнению прямой

$$y = \frac{y_2 - y_1}{x_2 - x_1} x + \frac{y_1 x_2 - y_2 x_1}{x_2 - x_1};$$

5) запишите условие, истинное, когда точка с координатами (x, y) лежит в заштрихованных областях рис. 2.5. Задания *а* – *п* соответствуют различным вариантам.



а *б* *в*

г *д* *е*

Рис. 2.5

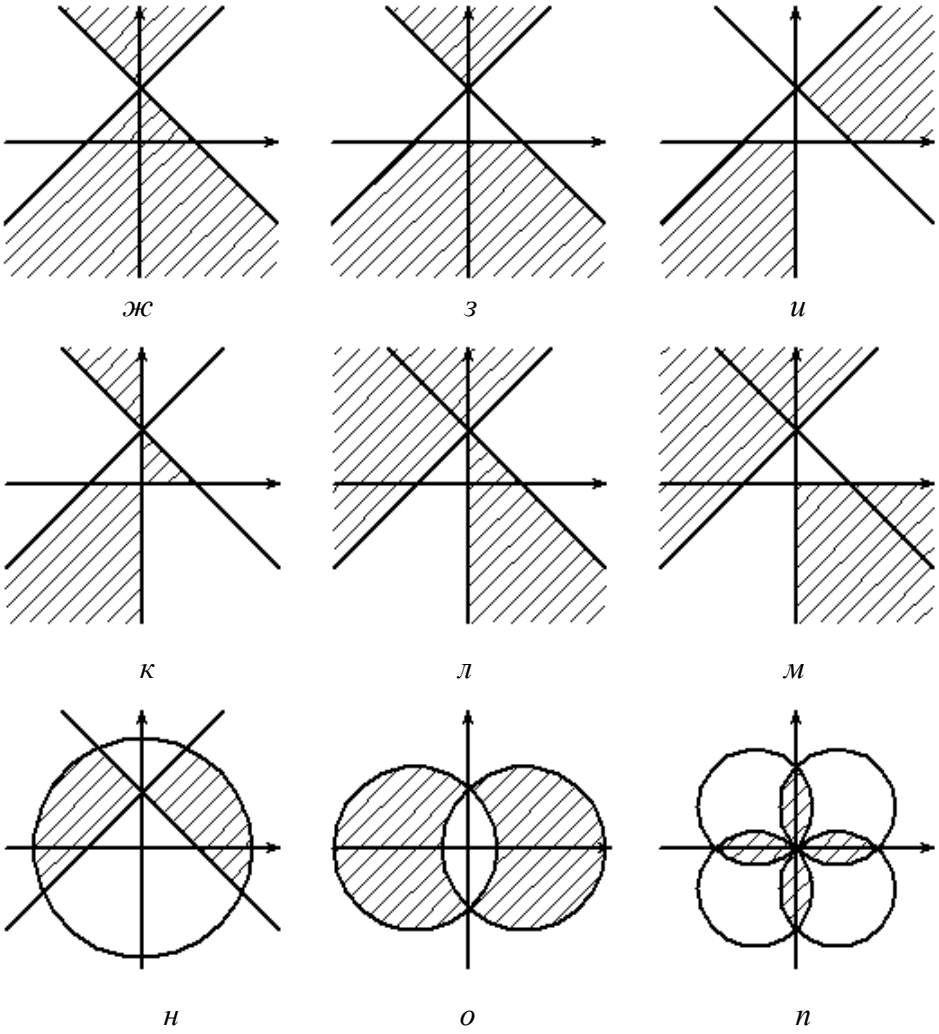


Рис. 2.5. Окончание

6) укажите на плоскости *XY* область, где истинными являются следующие логические выражения:

```
(abs(x - y) < 1) and (abs(x) + abs(y) > 1)
(abs(x - y) < 1) and (abs(x) + abs(y) > 1)
(abs(x) < 1) or (abs(y) < 1)
x^2 + y^2 > (x + y)^2
```

7) пусть A и B – логические выражения, принимающие значения true или false. Какие из приведенных пар составных логических выражений эквивалентны, т. е. при любых ли значениях A и B значения выражений слева и справа совпадают?

Указание. Формально проверить эквивалентность двух логических выражений можно, составив для них таблицы истинности:

а) **not** (A **and** B) и A **or** B
б) **not** (A **and** B) и (**not** A) **or** (**not** B)
в) **not** (A **or** B) и A **and** B
г) **not** (A **or** B) и (**not** A) **and** (**not** B)

```
д) (A and B) or                    и A = B
     ((not A) and (not B))
е) A <> B                          и ((not A) and B) or
                                        (A and (not B))
ж) A = B = true                    и A and B
з) A = B = true                    и A = B
и) A = false                       и false
к) (not (A or B)) and A            и (A or B) and (not B)
                                        and (not A)
л) (A or B) and (not B)            и A
```

8) зарплата выдается 5-го числа каждого месяца. Составьте логическое выражение, которое истинно, если на *k*-е число *m*-го месяца зарплата уже была выдана 10 раз с начала года.

2.5. Условный оператор

Условный оператор позволяет проверить некоторое условие и в зависимости от результатов проверки выполнить то или иное действие.

Например, создавая программу для решения квадратных уравнений, вы сталкивались с возможностью того, что при заданных пользователем коэффициентах дискриминант будет отрицательный. Чтобы программа могла правильно работать в любом случае, вычисление корней следует производить только при неотрицательности дискриминанта.

Условный оператор имеет следующую структуру:

```
if <условие – любое логическое выражение> then
begin
   <Операторы 1>
end else
begin
   <Операторы 2>
end;
```

if, **then**, **else** – зарезервированные слова (если, то, иначе).

Если условие имеет значение true, то выполняется 1-я группа операторов, иначе 2-я группа. Если при истинности (или не истинности) условия требуется выполнить всего один оператор, то слова **begin** и **end** можно опустить.

Пример. Программа, выбирающая меньшее число из двух веденных:

```
var
   x, y: integer;
begin
   readln(x, y);
   if x<y then
      writeln(x)
```

```
   else
      writeln(y);
end.
```

В примере видим как раз случай, когда слов **begin** и **end** нет. Иногда рекомендуют не думать о количестве операторов, а ставить **begin** и **end** всегда. Это позволит избежать частых ошибок из-за их отсутствия.

Обратите внимание на следующую особенность: *перед словом **else** не ставится точка с запятой.* Так, в примере нет точки с запятой после оператора writeln(x).

Есть известный программистский анекдот: программист ставит на ночь перед кроватью два стакана: один с водой (если проснется и захочет пить) и один пустой (если проснется, но пить не захочет). Чтобы избежать подобного абсурда, существует укороченная форма условного оператора:

```
if <условие> then
begin
   <операторы>
end;
```

При истинности условия <условие> выполнятся операторы <операторы>. Если же условие не выполняется, данный оператор не сделает ничего.

2.6. Оформление текста программ

Как известно, людям свойственно ошибаться. Особенно это верно в отношении программистов. По некоторым оценкам, ядро системы Linux содержит порядка 15000 пока еще не исправленных ошибок. Практически любой написанный вами текст программы будет содержать ошибки, и значительная часть времени будет уходить на их поиск и исправление.

Отсюда вытекает преставление о стиле программирования. Существуют определенные правила написания программ, соблюдение которых позволяет уменьшить вероятность появления ошибок. Программы (особенно такие простые, как вам придется писать на начальном этапе) могут правильно работать и при нарушении этих правил, однако их все же следует соблюдать. Дополнительное время и усилия, которые будут на это потрачены, многократно окупятся впоследствии.

Простейшее, но при этом очень важное стилистическое правило описывает то, как надо располагать текст программы. Существует несколько вариантов соглашений о правильном расположении текста. В данном пособии мы будем следовать Object Pascal Style Guide (см. http://edn.embarcadero.com/article/10280) – стандарту, выработанному разработчиками языка Delphi.

Итак, следует действовать следующим образом:

1) слово **var** пишется на отдельной строчке. Следующие за ним описания переменных начинаются с новой строки. При этом у всех описаний

делается отступ слева в два пробела. Переменные разных типов описываются на разных строках:

правильно:

```
var         {слово var на отдельной строке без отступа
               слева}
   x, y: real;   {описания переменных с новой строки.
               Отступ слева в два пробела}
   i: integer;{переменная другого типа на отдельной
               строке, с тем же отступом}
```

2) в полиграфии существует такое понятие, как типографика. Это набор правил, описывающих, как должен располагаться на страницах книги текст, чтобы быть удобочитаемым. Одно из простых правил говорит, что после знаков препинания всегда ставится пробел. Не следует пренебрегать этим правилом и при написании программ. Пробел всегда следует ставить после запятых и двоеточий (как в вышеприведенном примере);

3) слова **begin** и **end**, ограничивающие раздел операторов, пишутся без отступа. Весь же текст программы между ними снова пишется с отступом в два пробела;

4) на одной строке должен располагаться только один оператор. Так, хотя допустимо писать, например,

```
begin
   x:=1; y:=2; z:=3; writeln(x, y, z);
end.
```

следует все же писать

```
begin
   x:=1;
   y:=2;
   z:=3;
   writeln(x, y, z);
end.
```

5) условный оператор записывается следующим образом:

```
begin
   ...
   {Первая строка с отступом в два пробела}
   if <условие> then
   begin              {слово begin с тем же отступом,
                      что и слово if}
     <Операторы 1>    {отступ на два пробела больше,
                      чем у if и begin}
   end else           {end обязательно с тем же отсту-
                      пом, что и соответствующий begin}
```

```
begin
      <Операторы 2>  {отступ на два пробела больше,
                      чем у if и begin}
   end;
   …
end.
```

Пример правильного оформления:

```
var
   x, y: real;
begin
   readln(x, y);
   if x > y then
   begin
      writeln('Max(x, y) = ', x);
   end else
   begin
     writeln('Max(x, y) = ', y);
   end;
   readln;
end.
```

Пример неправильного оформления:

```
{Переменные на той же строке, что и var}
var x, y: real;
begin
   readln(x, y);
   {Оператор на той же строке, что и if}
   if x > y then writeln('Max(x, y) = ', x)
   else
      {begin с отступом относительно if}
      begin
      {writeln без отступа относительно begin}
      writeln('Max(x, y) = ', y);
      end;
{readln без отступа}
readln;
end.
```

Данная «неправильная» программа будет работать совершенно так же, как и приведенная выше «правильная», однако использовать надо именно правильный вариант;

6) внутри условного оператора могут располагаться любые другие операторы, в том числе и другие условные операторы. То, что находится внутри них, пишется с еще большим отступом.

Например:

```
begin
   readln(x, y, z);
   if x > y then
   begin
       if x > z then {вложенный if с дополнительным
                      отступом в два пробела}
       begin
          writeln(x);{writeln с еще большим отступом}
       end;
   end;
end.
```

Общая идея такова: отступы показывают подчиненность (вложенность) структур в вашей программе. Текст внутри раздела операторов: делаем отступ, внутри условного оператора: еще больший отступ, внутри еще одного условного оператора – еще больший отступ и т. д.

Соблюдение данных правил делает визуально очевидной логическую структуру вашей программы. Создание сколько-нибудь сложной программы без этого оказывается невозможным.

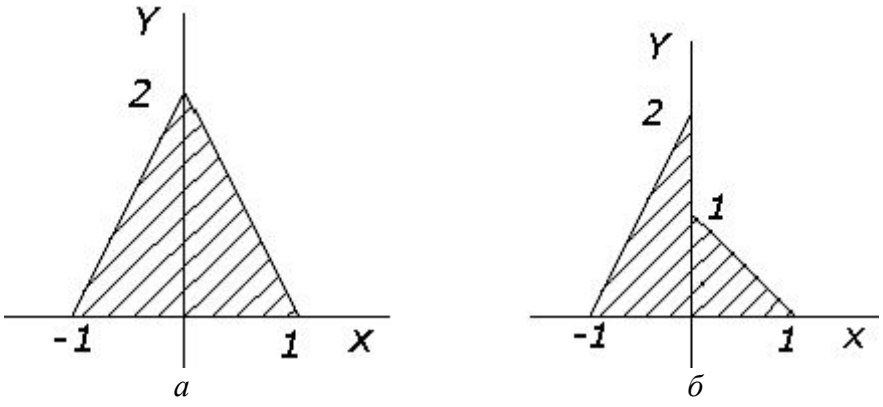
Контрольная работа 2

1. Вычислите, какое значение будет присвоено логической переменной *b*:

а)
```
x := 2;
y := 5;
b := not((x>=2)and(x*y<5));
```

б)
```
x := 2;
y := 55;
b := not(not(2*x>4)and(y mod 2<>1));
```

2. Составьте логическое выражение, которое истинно, когда точка с координатами (*x*, *y*) попадает в заштрихованную область на рис. 2.6:



а *б*

Рис. 2.6

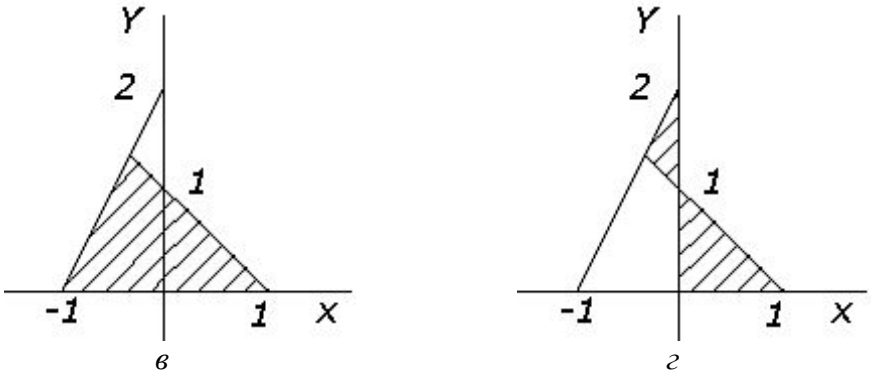


Рис. 2.6. Окончание

3. Какое значение примут переменные после выполнения следующих операторов:

а)
```
x := 5;
y := 10;
if sqr(x) > sqrt(y)
then
   x:=sqr(x)
else
   y:= sqr(y);
```

б)
```
x := 5;
y := 10;
if sqr(x) > sqrt(y) then
   x := sqr(x)
else
   y := sqr(y);
   x := sqr(x);
```

Задание 2. Составление логических выражений, условный оператор

1. Напишите программу, которая запрашивает два числа, а затем выводит их в порядке возрастания – сначала меньшее, затем большее.

2. Создайте программу, которая запрашивает у пользователя три числа, а затем сообщает ему, какое из этих чисел наибольшее.

3. Создайте программу, которая запрашивает у пользователя число и сообщает, является ли оно число четным.

4. Даны три числа – a, b, c. Если среди них есть отрицательные, возведите их в квадрат. Если после возведения в квадрат число стало больше 20, умножьте его на 2.

5. Напишите программу, которая запрашивает значение x, а затем выводит значение следующей функции от x:

$$\operatorname{sign}(x) = \begin{cases} 1, & x > 0; \\ 0, & x = 0; \\ -1, & x < 0. \end{cases}$$

6. Напишите программу, которая запрашивает значения x, y, z, а затем выводит значение следующих функций:

$$\max(x, \min(y, z)) \text{ и } \min(\min(x, y), z).$$

7. Модифицируйте вашу программу расчета корней квадратного уравнения, добавив к ней проверку неотрицательности дискриминанта.

Если дискриминант отрицательный, сообщайте пользователю, что уравнение не имеет корней.

8. Модифицируйте вашу программу расчета корней квадратного уравнения, добавив к ней проверку того, что первый коэффициент не равен нулю. В противном случае сообщайте пользователю, что уравнение не квадратное, а линейное, и вычислите его единственный корень. Если первые два коэффициента равны нулю, а третий не равен, сообщите пользователю, что корней нет. А если все коэффициенты равны нулю, сообщите, что любое число является корнем.

9. Пользователь вводит три числа. Сообщите ему, упорядочены ли введенные числа по возрастанию.

10. Пользователь вводит три числа – длины сторон треугольника. Программа должна сообщить пользователю, каким является треугольник:

– равносторонним;
– равнобедренным;
– разносторонним;
– прямоугольным;

а также существует ли вообще такой треугольник (такого треугольника не может быть, если сумма любых двух сторон окажется меньше третьей стороны).

11. «Узник замка Иф».

За многие годы заточения узник замка Иф проделал вилкой в стене прямоугольное отверстие размером $d \times e$. Замок Иф сложен из кирпичей размером $a \times b \times c$. Узник хочет узнать, сможет ли он выбрасывать кирпичи в море из этого отверстия, чтобы сделать подкоп. Снабдите его необходимым для решения задачи софтом. На вход программе подаются 5 чисел (a, b, c, d, e), программа должна давать ответ YES или NO.

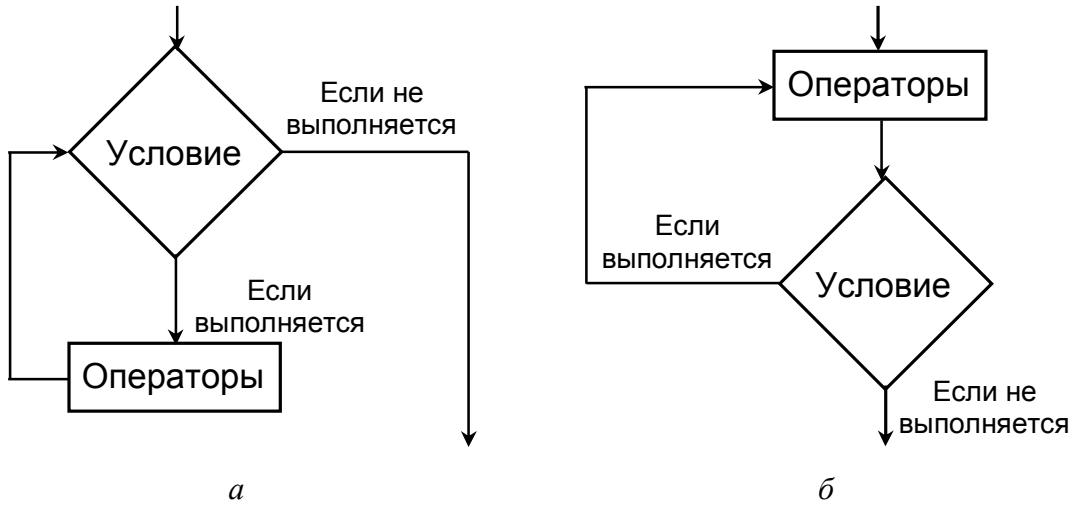
12. Напишите программу, которая в зависимости от введенного возраста добавляет слова «год», «года» или «лет». Например, при вводе возраста 1, программа сообщает «1 год», при числе 2 – «2 года», при числе 125 – «125 лет».

3. Цикл *for*

3.1. Цикл с параметром (*for*)

Современные компьютеры выполняют миллиарды операций в секунду. Однако, к счастью, программистам не приходится писать программы из миллиардов строк кода. Чтобы с помощью короткой программы указать на необходимость выполнения большого количества операций, используется оператор цикла, суть которого в многократном повторении одного и того же набора инструкций. Работа циклов с помощью блок-схем показана на рис. 3.1. В варианте (*a*), если «условие» истинно, выполняются «операторы» (см. рис. 3.1). После этого снова проверяется «условие», и если оно по-прежнему истинно, то снова выполняются операторы и т. д., пока условие

не станет ложным. Вариант (*б*) подобен варианту (*а*) (см. рис. 3.1), отличие только в том, что сначала выполняются «операторы» и только затем делается проверка «условия», на основе которой мы решаем, нужно ли повторять их выполнение.



а *б*

Рис. 3.1

В языке Паскаль существует несколько вариантов организации циклов. Рассмотрим один из них, так называемый *цикл с параметром* или *цикл for*. Чтобы записать его, нам потребуется переменная целого типа, например:

```
var
   i: integer;
```

Схема его записи выглядит следующим образом:

```
for i:=<начальное значение> to <конечное значение> do
begin
   <операторы>
end;
```

Здесь i – так называемая *переменная-счетчик* (разумеется, ее не обязательно называть именно i, это может быть любая переменная целого типа). Начальное и конечное значения – это любые выражения, имеющие значение целого типа.

Когда оператор цикла начинает работу, переменная-счетчик принимает начальное значение, затем выполняются <операторы>, после этого счетчик увеличивается на единицу и снова выполняются <операторы>. Процесс повторяется, пока счетчик не окажется больше конечного значения. Например, если начальное значение 2, а конечное 3, то <операторы> будут выполнены 2 раза.

Область между словами **begin** и **end**, где располагаются повторяющиеся в цикле операторы, называется *телом цикла*.

Блок-схема работы этого цикла показана на рис. 3.2.

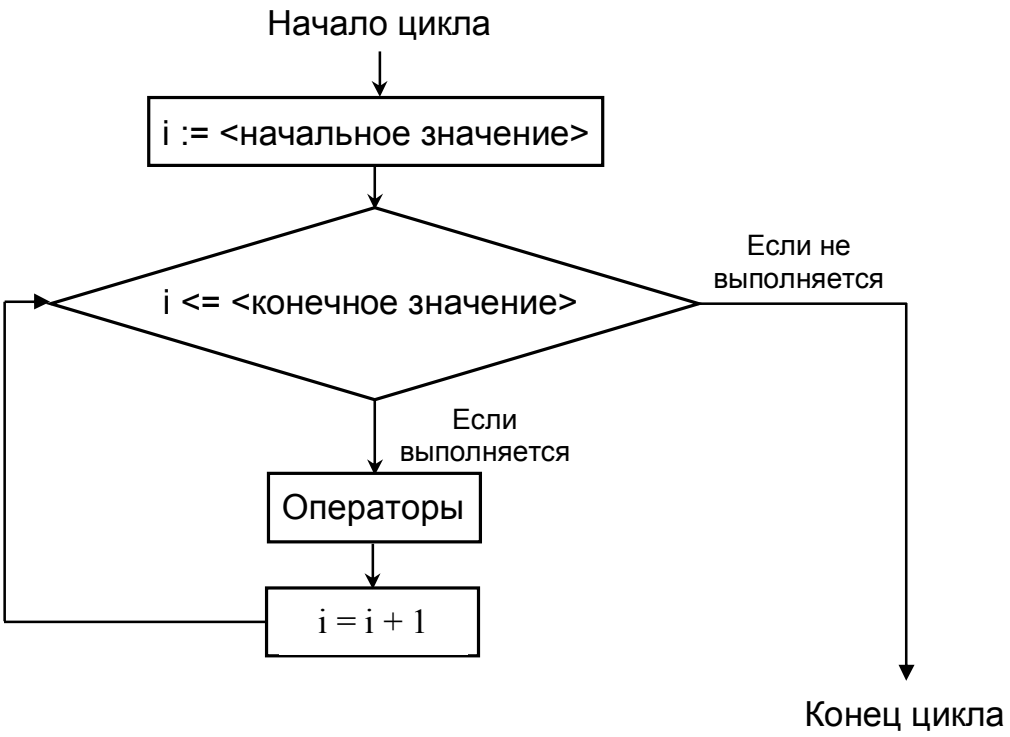


Рис. 3.2

Пример 1. Напечатать слово Hello на экране 100 раз.

Один раз слово Hello можно напечатать с помощью процедуры writeln('Hello'). Чтобы напечатать его 100 раз, надо эту инструкцию поместить внутрь оператора цикла, который выполнится нужное количество раз. А именно:

```
for n:=1 to 100 do
begin
   writeln('Hello');
end;
```

Переменной счетчику *n* будет присвоено начальное значение 1. Затем Hello будет напечатано 1-й раз. Счетчик увеличится на 1 и Hello напечатается 2-й раз и т.д.

Перечислим в одном месте все *правила, касающиеся работы цикла с параметром*:

1) переменная-счетчик должна быть обязательно целого типа (например, integer);

2) начальное и конечное значения задаются выражениями, значение которых также имеет целый тип. Нельзя, например, записать цикл

```
for x:=0 to 2*Pi do …
```

Но можно, например, так:

```
for k:=2*round(n*1000) to trunc(2*pi)+1 do …
```

3) если конечное значение меньше начального, цикл не выполнится ни разу;

4) после окончания работы переменная-счетчик «портится». Глядя на блок-схему, можно предположить, что после окончания работы цикла она станет на единицу больше конечного значения. На практике это не так, и она может иметь любое значение. Отсюда правило: если переменная используется как счетчик шагов в цикле, не следует обращаться к ней после того, как цикл завершил свою работу;

5) если в теле цикла содержится всего один оператор, то слова **begin** и **end**, ограничивающие тело цикла, можно опустить. Такая ситуация у нас была в примере 1. Там можно было написать:

```
for n:=1 to 100 do
   writeln('Hello');
```

6) **важное стилистическое правило**: хотя это и не запрещено, не следует в теле цикла использовать какие-либо операторы, меняющие значения переменной-счетчика. В частности, нельзя ей ничего присваивать. Нарушение этого правила ведет к увеличению вероятности сделать трудно обнаруживаемую ошибку. Соблюдение – никак не ограничивает ваши возможности;

7) тело цикла может содержать любые операторы. Следовательно, туда можно поместить другой оператор цикла. Переменные-счетчики в этом случае у циклов должны быть разные. Если их сделать одинаковыми, это нарушит предыдущее правило – внутренний цикл изменит значение переменной-счетчика внешнего;

8) еще одно стилистическое правило: все операторы тела цикла должны быть дополнительно сдвинуты на 2 – 3 пробела вправо (см. запись программы в примере 1 и последующих примерах ниже по тексту).

Итак, одни и те же действия мы можем выполнять сколько угодно раз, но что полезного это дает? Печатать 100 раз Hello – не очень важное приложение. Надо сделать так, чтобы действия на каждом шаге цикла чем-то различались, тогда это будет полезное дело. Первый способ добиться этого состоит в использовании переменной-счетчика, которая на каждом шаге принимает различные значения.

Пример 2. Напечатать все целые числа в диапазоне от 5 до 25.

```
for i:=5 to 25 do
   writeln(i);
```

Как видно, в данном примере на каждом шаге печатаются разные числа. Используем эту возможность для чего-то более полезного.

Пример 3. Табуляция функций.

Под табуляцией функций подразумевается составление таблицы значений функции для некоторого диапазона значений аргумента. Пусть требуется `N` значений функции `f(x)` в диапазоне от `Xmin` до `Xmax`. Рассмат-

ривая переменную-счетчик как номер аргумента функции, составим выражение, позволяющее по номеру получить значение аргумента:

```
x := Xmin + (Xmax - Xmin)*(i-1)/(N-1).
```

Убедитесь, что действительно первое значение аргумента (при `i = 1`) составляет `x = Xmin`, а `N-е (i = N)` – `x = Xmax`. Вычисляя на каждом шаге значение аргумента, можем тут же вычислить функцию и составить требуемую таблицу.

```
for i:=1 to N do
begin
   x:=Xmin+(Xmax-Xmin)*(i-1)/(N-1);
   y:=f(x);
   writeln(x, '   ', y);
end;
```

Вместо `f(x)` в приведенной программе следует подставить любое выражение, составленное по правилам языка Паскаль.

3.2. Прием накопления суммы

Данный алгоритмический прием используется, когда надо просуммировать большое количество чисел. Для этого переменной, в которую будет записываться сумма, в начале присваивается нулевое значение, затем делается цикл, где на каждом шаге к этой переменной добавляется очередное число.

Пример 4. Просуммировать все целые числа от 1 до 100.

```
s:=0;           {обнуление переменной}
for i:=1 to 100 do
   s:=s+i;      {прибавление очередного элемента суммы
                 на каждом шаге цикла}
```

Очень важная, фундаментальная идея, использованная в данном приеме, состоит в том, что результат выполнения каждого шага цикла зависит от значения переменной, вычисленной на предыдущем шаге. Таким образом, вместо тривиального повторения одного и того же мы на каждом шаге получаем новый результат. Так, в приведенном примере очередное число добавляется к значению переменной s, полученному на предыдущем шаге.

А к чему добавляется очередное число на самом первом шаге? Чтобы было к чему добавлять, перед циклом обязательно должна присутствовать инициализация (присваивание начального значения) переменной, в которой накапливается сумма. Чаще всего требуется присвоить ей начальное значение 0.

Программистский анекдот в тему. Буратино подарили три яблока. Два он съел. Сколько яблок осталось у Буратино? Ответ «одно» непра-

вильный. Неизвестно, сколько осталось, так как не сказано, сколько яблок было у него до того, как ему подарили три новых. Мораль: ***не забывайте обнулять переменные***.

3.3. Прием накопления произведения

Аналогично накоплению суммы можно в отдельной переменной накапливать произведение. Переменной, в которой производится накопление, присваивается начальное значение 1.

Пример 5. Вычисление факториала.

Факториалом целого числа *n* называется произведение всех целых чисел от 1 до *n*. Обозначается *n*!. Иными словами:

$$n! = 1*2*3*\ldots*n$$

Вычисляющая факториал программа выглядит так:

```
readln(n);
p:=1;
for k:=2 to n do
   p:=p*k;
writeln(p);
```

3.4. Комбинация обоих приемов

Пример 6. Вычислить значение выражения 1!+2!+3!+…+*n*!

Решение в лоб состоит в том, чтобы в теле цикла, осуществляющего суммирование, производить вычисление факториала:

```
s:=0;
for i:=1 to n do
begin
   {Вычисление факториала от i}
   p:=1;
   for k:=1 to i do
      p:=p*k;
   {Добавление вычисленного факториала к сумме}
   s:=s+p;
end;
```

Заметим, однако, что при вычислении факториала на каждом шаге получается факториал все большего целого числа. Эти «промежуточные» результаты однократного вычисления факториала и можно суммировать:

```
s:=0;
p:=1;
for i:=1 to n do
begin
   p:=p*i;
   s:=s+p;
end;
```

3.5. Цикл с *downto*

Если вместо слова **to** в цикле **for** поставить **downto**, то после выполнения каждого шага цикла переменная-счетчик будет не увеличиваться, а уменьшаться на единицу. Так, приведенный ниже код

```
for i:=10 downto 1 do
   writeln(i);
```

печатает числа 10, 9, 8, …

Если начальное значение в цикле с **downto** будет меньше конечного, то тело цикла не выполнится ни разу.

3.6. Операторы *break* и *continue*

Ходом выполнения цикла можно управлять с помощью двух операторов **break** и **continue**.

break прерывает выполнение цикла, управление передается операторам, следующим за оператором цикла.

continue прерывает выполнение очередного шага цикла и возвращает управление в начало цикла, начиная следующий шаг.

Например:

```
for n:=1 to 10 do
begin
   if n mod 2 = 0 then
      continue;
   if n = 7 then
      break;
   writeln(n);
end;
```

Данная программа будет печатать только нечетные числа (из-за срабатывания continue). Цикл прекратит выполняться, когда n станет равно 7. В итоге будут напечатаны числа 1, 3, 5.

Контрольная работа 3

1. Сколько раз на экран выведется слово Hello?

```
for i:=-2 to 7 do
   writeln('Hello');
```

2. Какого типа должна быть переменная i в предыдущем задании?

3. Что изменится, если убрать операторные скобки (слова **begin** и **end**)?

```
for i:=1 to 10 do
begin
   writeln(i+1);
end;
```

4. Что выведут на экран приведенные программы?

а)
```
s:=1;
for i:=1 to 5 do
   if i>2 then
     s:=s*i;
writeln(s);
```

б)
```
s:=0;
a:=3;
for i:=1 to 10 do
   s:=a+i;
writeln(s);
```

в)
```
s:=0;
a:=5;
for i:=1 to 3 do
   s:=2*a;
writeln(s);
```

5. Составьте таблицы изменения переменных для циклов

а)
```
s:=0;
p:=1;
for i:=1 to 5 do
begin
   p:=p*i;
   s:=s+p;
end;
```

б)
```
c:=2;
for i:=1 to 4 do
   c:=1/(1-c);
```

в)
```
s:=0;
p:=1;
for i:=1 to 5 do
begin
   p:=-p*2;
   s:=s+p;
end;
```

Задание 3. Цикл *for*. Приемы накопления суммы и произведения

1. Цикл *for*

1.1. Напечатайте таблицу умножения на 5, желательно печатать
1 x 5 = 5,
2 x 5 = 10,
…
а не просто 5, 10 и т.д.

1.2. Напечатайте в столбик нечетные числа от 3 до 25.

1.3. Напечатайте свое имя в углах экрана.

1.4. Выведите на экран таблицу значений синуса от 0 до 2π. В каждой строке должны стоять один аргумент и одно значение. Количество значений аргумента пусть задает пользователь.

2. Прием накопления суммы

2.1. Напишите программу, которая вычисляет сумму квадратов чисел от 1 до *N*. Число *N* программа должна запрашивать у пользователя.

2.2. Напишите программу, перемножающую целые числа без использования операции умножения. Например, при умножении целых чисел $n \cdot m$ число *m* надо сложить само с собой *n* раз (*m*+*m*+…+*m*).

2.3. Используя прием накопления суммы, найдите сумму нечетных чисел от 1 до *N*. Число *N* программа должна запрашивать у пользователя.

2.4. Выведите на экран последовательность сумм чисел от 1 до *n*; *n* меняется от 1 до 10. Иными сорвами, первые члены последовательности – 1, 3 (1+2), 6 (1+2+3), 10 (1+2+3+4) и т.д.

2.5. Вычислите сумму элементов последовательности сумм из предыдущего задания.

2.6. Найдите сумму 100 синусов от аргументов в диапазоне от 0 до 2π.

3. Прием накопления произведения

3.1. Факториалом целого числа *n* (обозначается *n*!) называется произведение всех целых чисел от 1 до *n*. Напишите программу вычисления факториала введенного пользователем числа.

3.2. Напишите программу возведения числа в целую степень. Число и степень запрашивайте у пользователя.

4. Комбинация обоих приемов

4.1. Используя комбинацию обоих приемов, напишите программу, вычисляющую функцию $1 + x + x^2 + \ldots + x^{10}$.

4.2. Вычислите сумму ряда $s_i = 1 + x + \frac{x^2}{2!} + \ldots + \frac{x^i}{i!}$.

Из математического анализа известно, что $\lim\limits_{n\to\infty} s_n = \exp(x)$. Выясните, насколько 5-й и 10-й члены последовательности таких сумм отличаются от exp(*x*).

4. Вычисления с помощью рекуррентных соотношений

4.1. Рекуррентные соотношения: основные понятия

В основе рассмотренных ранее алгоритмических приемов накопления суммы и произведения лежит фундаментальная идея о том, что результат вычислений на каждом шаге цикла должен зависеть от результата вы-

числений на предыдущем шаге. Обобщенным математическим выражением этой идеи являются *рекуррентные соотношения*.

Будем говорить, что последовательность векторов $\{\vec{x}_n\}$ задана рекуррентным соотношением, если заданы начальный вектор $\vec{x}_0 = (x_0^1, x_0^2, \ldots, x_0^D)$ и функциональная зависимость последующего вектора от предыдущего

$$\vec{x}_{n+1} = \bar{f}(\vec{x}_n). \qquad (1)$$

Вектора $\vec{x}_n$ можно интерпретировать как наборы значений переменных. Таким образом, они характеризуют *состояние* вычислительного процесса. Функцию $\bar{f}$ будем понимать как преобразование значений переменных на каждом шаге цикла.

Задача. Пусть задано рекуррентное соотношение

$$x_{n+1} = 2 - x_n^2,$$

начальное значение $x_0 = 0$. Найдите x_5.

Пример 1. Запишите рекуррентные соотношения для нахождения суммы целых чисел от 1 до *m* и факториала *m*!

Для суммы начальное значение $x_0 = 0$, рекуррентное соотношение $x_{n+1} = x_n + n$. Требуемая сумма будет найдена на *m*-м шаге.

Аналогично для факториала: $x_0 = 1$, $x_{n+1} = n \cdot x_n$, требуемое значение также будет найдено на *m*-м шаге.

Как видно, вычислительные процессы, соответствующие накоплению суммы и произведения, действительно могут быть заданы рекуррентными соотношениями.

В приведенных примерах рекуррентные соотношения явно содержали номер шага *n*, чего, вообще говоря, нет в формуле (1). С практической точки зрения это не важно – в цикле **for** на каждом шаге можно использовать значение переменной-счетчика шагов. Однако, заботясь о математической строгости, нетрудно свести все к виду (1), сделав номер шага элементом вектора состояния вычислительного процесса. Так, для вычисления факториала будем иметь

$$\begin{cases} x_{n+1} = y_n \cdot x_n, \\ y_{n+1} = y_n + 1 \end{cases}$$

при начальных условиях

$$x_0 = 1,\ y_0 = 1.$$

Однократное вычисление следующих значений по предыдущим посредством рекуррентных соотношений называется *итерацией*. А процесс вычислений с помощью рекуррентных соотношений соответственно *итерированием*.

4.2. Задачи на составление рекуррентных соотношений

1. Придумайте рекуррентное соотношение, задающее следующие числовые последовательности:

а) 1, 2, 3, 4, …

б) 0, 5, 10, 15, …

в) 1, 1, 1, 1, …

г) 1, –1, 1, –1, …

д) 1, –2, 3, –4, 5, –6, …

е) 2, 4, 8, 16, …

ж) 2, 4, 16, 256, …

з) 0, 1, 2, 3, 0, 1, 2, 3, 0, …

и) 1!, 3!, 5!, 7!, …

2. Придумайте рекуррентные соотношения для вычисления последовательностей следующего вида:

а) $1,\ a,\ a^2,\ a^3,\ a^4,\ \ldots$

б) $1,\ \frac{a}{1!},\ \frac{a^2}{2!},\ \frac{a^3}{3!},\ \ldots$

в) $1,\ 1+a,\ 1+a+a^2,\ 1+a+a^2+a^3,\ \ldots$

3. Придумайте рекуррентное соотношение, укажите начальное значение и число шагов, необходимые для вычисления следующих величин:

а) a^{m+1}

б) $2a^m$

в) $(a-1)^m$

г) $\frac{a^m}{(a-1)^{m+1}}$

д) $(2m-1)!$

е) $1+a+a^2+\ldots+a^k$

ж) $1+a^2+a^4+\ldots+a^{2k}$

з) $1-a+a^2-a^3+\ldots-a^{2m-1}$

и) $1+x+\frac{x^2}{2!}+\frac{x^3}{3!}+\ldots+\frac{x^m}{m!}$

к) $x-\frac{x^3}{3!}+\frac{x^5}{5!}-\ldots+(-1)^{m-1}\frac{x^{2m-1}}{(2m-1)!}$

4. Составьте рекуррентные соотношения для приведенных ниже величин.

Указание. Как видно, данные формулы основаны на повторении одной или нескольких операций. Рекуррентное соотношение должно содержать их однократное выполнение:

а) золотое сечение

$$1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+...}}}.$$

Если вычисления продолжать до бесконечности, то результат сойдется к числу, называемому золотое сечение (оно же золотая пропорция, гармоническое деление и число Фидия). Данное число было известно еще до нашей эры (его открытие приписывают Пифагору), а само название было дано Леонардо да Винчи. Считается, что объекты, в которых присутствуют пропорции, равные этому числу, воспринимаются как красивые;

б) $\sqrt{2}=1+\cfrac{1}{2+\cfrac{1}{2+\cfrac{1}{2+...}}}$; в) $\sqrt{2+\sqrt{4+\sqrt{6+...+\sqrt{98}}}}$.

4.3. Многомерные рекуррентные соотношения

Размерностью рекуррентного соотношения называют размерность (количество компонент) вектора состояния $\vec{x}_n$. Соответственно говорят об одномерных или многомерных рекуррентных соотношениях. Программирование вычислений с помощью одномерного рекуррентного соотношения заключается в простом помещении его внутрь цикла. Например, если надо найти 10-й член последовательности $x_{n+1}=2-x_n^2$ при $x_0=0.5$:

```
x:=0.5;
for i:=1 to 10 do
   x:=2-x*x;
```

Если переменных несколько, возникает небольшая тонкость. Для примера рассмотрим двумерное соотношение общего вида

$$\begin{cases} x_{n+1}=f(x_n,y_n), \\ y_{n+1}=g(x_n,y_n). \end{cases} \qquad (2)$$

Если снова поместить эти формулы внутрь цикла, как показано ниже:

```
for i:=1 to 10 do
begin
   x:=f(x, y);
   y:=g(x, y);
end;
```

получится, что строчка

```
x:=f(x, y);
```

запишет в x значение x_{i+1}, которое будет использовано в следующей строке вместо требуемого x_i. Таким образом, чтобы корректно итерировать многомерные рекуррентные соотношения, надо перед вычислением очередного члена последовательности запоминать предыдущее значение. Для двумерного рекуррентного соотношения (2) корректная программа будет выглядеть как

```
for i:=1 to 10 do
begin
   x2:=x;      {запоминаем последний из вычисленных
                членов последовательности – x_i }
   x:=f(x, y); {вычисляем следующий элемент x_{i+1} }
   y:=g(x2, y);  {для вычислений используем запомнен-
                ное значение x_i }
end;
```

Другой способ заключается том, что в дополнительные переменные записываются новые члены последовательности. Для двумерного соотношения это будет выглядеть так:

```
for i:=1 to 10 do
begin
   x2:=f(x, y);  {новый член последовательности запо-
                  минается в дополнительной переменной
                  x2}
   y:=g(x, y);
   x:=x2;
end;
```

Контрольная работа 4

1. Укажите 4-й член последовательности, заданной рекуррентным соотношением

$$x_{n+1} = 2 - \frac{1}{x_n},\ x_1 = 2.$$

2. Какими рекуррентными соотношениями задаются последовательности?

а) 2, 4, 16, 256, …

б) 2, 0.5, 2, 0.5, 2, …

в) 2, 5, 8, 11, 14, …

г) 2, -4, 16, -256, …

3. Какую функцию переменной x вычисляет программа? Укажите также рекуррентные соотношения, в соответствии с которыми происходят вычисления:

а)
```
readln(x);
y:=0;
p:=1;
sgn:=1;
for i:=2 to 4 do
begin
   p:=p*x;
   y:=y+sgn*p;
   sgn:=-sgn;
end;
writeln(y);
```

б)
```
readln(x);
y:=0;
p:=1;
for i:=1 to 4 do
begin
   p:=-p*x*x;
   y:=y+p/i;
end;
writeln(y);
```

в)
```
readln(x);
y:=1;
p:=1;
for i:=1 to 4 do
begin
   p:=p*(x-1);
   y:=y+p;
end;
writeln(y);
```

г)
```
readln(x);
y:=1;
p:=1;
for i:=1 to 4 do
begin
   p:=-p/x;
   y:=y+p;
end;
writeln(y);
```

Контрольная работа 5

1. Каким рекуррентным соотношением описывается последовательность?

$$0, \frac{1}{2}, \frac{3}{4}, \frac{7}{8}, \ldots$$

2. Запишите рекуррентное соотношение и первый член последовательности, необходимые для вычисления величины

а) $\dfrac{a^{n-2}}{(a-1)^n}$,

б) $(a-1)^{2n+1}\left(\dfrac{a}{2}\right)^{2n-1}$.

3. Запишите рекуррентные соотношения, необходимые для вычисления функции:

а) $y = 1 - x^3 + x^6 - x^9 + \ldots$

б) $y = 1 - x^2 - x^4 - x^6 - x^8 - \ldots$

в) $y = \dfrac{1}{x} + \dfrac{2}{x^2} + \dfrac{3}{x^3} + \dfrac{4}{x^4} + \ldots$

г) $y = \dfrac{1!}{x} + \dfrac{2!}{x} + \dfrac{3!}{x} + \dfrac{4!}{x} + \ldots$

4. Какую функцию переменной *x* вычисляет программа?

а)
```
readln(x);
y:=1;
p:=1;
for i:=-2 to 2 do
begin
   if i<>0 then
     p:=p*x*i/abs(i);
   y:=y+p;
end;
writeln(y);
```

б)
```
readln(x);
y:=1;
for i:=0 to 3 do
begin
  x:=1+1/x;
  y:=y-x;
end;
writeln(y);
```

5. Имеется двумерное рекуррентное соотношение

$$\begin{cases} x_{n+1} = x_n + y_n, \\ y_{n+1} = y_n - x_n. \end{cases}$$

Начальные условия $x_1 = 1$, $y_1 = 1$. Напишите программу, которая найдет x_{20} и y_{20}.

Задание 4. Вычисления с помощью рекуррентных соотношений

1. Последовательность определяется соотношением $x_{n+1} = \lambda - x_n^2$, где $\lambda = 2$, $x_0 = 1/\sqrt{2}$. Найти x_{10}, x_{20} и x_{30}.

2. Вычислите золотое сечение по формуле

$$1 + \cfrac{1}{1 + \cfrac{1}{1 + \ldots}}.$$

Сделайте 20 шагов. Определите, на сколько точнее вы узнаете золотое сечение, если сделать 30 шагов.

3. В 1674 году Г. Лейбниц показал, что число $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$ Найдите приближенное значение числа π, просуммировав 100 членов этого ряда.

4. Составив соответствующие рекуррентные соотношение, вычислите значения следующих выражений:

1) $1 + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{n!}$;

2) $n + \frac{n-1}{2!} + \frac{n-2}{3!} + \ldots + \frac{1}{n!}$;

3) $\frac{(x-2)(x-4)\ldots(x-2n)}{(x-1)(x-3)\ldots(x-2n+1)}$;

4) $\underbrace{\sqrt{2+\sqrt{2+...+\sqrt{2+\sqrt{2}}}}}_{n\text{ раз}}$;

5) $\sin x+\sin x^2+...+\sin x^n$;

6) $\sin x+\sin(\sin x)+...+\underbrace{\sin(\sin(\sin(...)))}_{n\text{ раз}}$.

5. Пользователь вводит 10 чисел. Определите, образуют ли они возрастающую последовательность.

5. Вложенные циклы

5.1. Вложенные циклы: теория

Циклы позволяют повторять выполнение любого набора операторов. В частности, можно повторять много раз выполнение другого цикла. Такие циклы называются вложенными.

Пример 1. Напечатать числа в виде следующей таблицы:

```
3 3 3 3 3
3 3 3 3 3
3 3 3 3 3
3 3 3 3 3
```

Данная таблица состоит из четырех строчек, в каждой из которых число 3 напечатано 5 раз. Строчку из пяти чисел можно напечатать с помощью одного цикла **for**:

```
for i:=1 to 5 do
   write(3, '   ');
```

Чтобы повторить вывод строчки 4 раза, вставляем этот цикл внутрь другого:

```
for k:=1 to 4 do
{4 раза делаем то, что написано между begin'ом и
end'ом}
begin
   for i:=1 to 5 do
      write(3, '   ');    {выводим одну строку}
   writeln;  {переводим курсор на следующую строку}
end;
```

Типичная ошибка, когда в качестве счетчиков вложенных циклов (i и k в приведенном примере) используется одна и та же переменная. Иными словами, нельзя в каждом из циклов использовать одну переменную i. Помнить об этом особенно важно, поскольку данная ошибка не обнаруживается на этапе компиляции. Ваша программа запустится, но делать будет

вовсе не то, что вы от нее ждете. В приведенном примере (если допустить ошибку, заменив переменную k на i) внешний цикл выполнится всего 1 раз вместо 4. Возможна также ситуация, когда такая ошибка приведет к зацикливанию: внешний цикл будет выполняться бесконечно долго – программа зависнет.

Рассмотрим еще один пример.

Пример 2. Напечатайте числа в виде следующей таблицы:

```
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 16
```

Снова используем внешний цикл для вывода строк, а внутренний для отдельных чисел в одной строке. Используем также отдельную переменную-счетчик n, в которой будет храниться выводимое число:

```
n:=1;
for i:=1 to 4 do
begin
   for k:=1 to 4 do
   begin
      write(n, '   ');
      n:=n+1;
   end;
   writeln;
end;
```

Дополнительная переменная-счетчик (n) здесь введена для большей прозрачности алгоритма. Заметив, что всегда выполняется $n=(i-1)\cdot 4+k$, можно обойтись без нее.

Разумеется, ту же задачу можно решить без вложенных циклов (вообще говоря, обойтись без них можно всегда). Например, заметив, что числа, стоящие в конце каждой строки, делятся на 4, делаем так:

```
for n:=1 to 16 do
begin
   write(n, '   ');
   if n mod 4 = 0 then
      writeln;
end;
```

Однако решения, основанные на вложенных циклах, как правило, интуитивно понятнее, их проще придумывать. Это связано с такой фундаментальной идеей программирования, как разделение задачи на подзадачи. Выделяется отдельная часть решения задачи (вывод одной строки в приведенных примерах), затем с помощью внешнего цикла она выполняется много раз.

Напомним, что в качестве пределов изменения переменной-счетчика цикла можно использовать произвольное выражение, дающее результат целого типа:

for i:=<произвольное выражение> **to** <произвольное выражение> **do**
…

В частности, если речь идет о вложенных циклах, то пределы изменения переменной во внутреннем цикле могут зависеть от значения переменной внешнего цикла.

Пример 3. Напечатайте числа в виде следующей таблицы:

1 2 3 4 5
3 4 5 6 7
5 6 7 8 9
7 8 9 10 11

Решение:

```
for i:=1 to 4 do
begin
   for k:=2*i-1 to (2*i-1)+4 do
      write(k, '   ');
   writeln;
end;
```

Поскольку внутри цикла может находиться все, что угодно, то ничто не мешает разместить там два цикла. Например, так:

```
for i:=1 to 10 do
begin
   …
   for k:=1 to 10 do
   begin
      ...
   end;
   for n:=1 to 10 do
   begin
      ...
   end;
   …
end;
```

Или так:

```
for i:=1 to 10 do
   for k:=1 to 10 do
      for n:=1 to 10 do …
```

В заключение небольшое замечание, касающееся правильного стиля написания программы, содержащей множество циклов (в частности, вло-

женных). Следует избегать одновременного использования в качестве счетчиков пар переменных с именами i и j, а также p и q. На вид они очень похожи, что часто приводит к трудно обнаруживаемым ошибкам.

Контрольная работа 6

1. Какое значение примет переменная *x* после выполнения программ:

а)
```
x:=0;
for i:=1 to 10 do
   for k:=i+1 to 10
do
      x:=x+1;
```

б)
```
x:=0;
for i:=1 to 10 do
   for k:=i+1 to 10-i do
      x:=x+1;
```

в)
```
x:=0;
for i:=1 to 5 do
   for k:=i-1 to i+1
do
      x:=x+k;
```

г)
```
x:=10;
for i:=1 to 5 do
   for k:=0 to i do
      x:=x+(k-i);
```

2. Что выведут программы?

а)
```
for i:=1 to 4 do
  begin
     if i mod 2 = 0
     then
        n:=i+1
     else
        n:=i;
     for k:=1 to n do
           write(n-i,
'   ');
        writeln;
     end;
```

б)
```
for i:=1 to 3 do
     for k:=3 downto 1 do
        for n:=i-k to (i+k)
  div 2 do
           write(n, '   ');
```

Задание 5. Вложенные циклы

1. Из математического анализа известно, что последовательность сумм вида

$$s_n = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + (-1)^{n+1}\frac{x^{2n-1}}{(2n-1)!}$$

сходится к функции $\sin x$. Исследуйте зависимость от x точности приближения синуса десятым членом последовательности. Для этого рассчитайте величину $\left|\sin x - s_{10}(x)\right|$ в 20 точках в диапазоне от 0 до 2π.

2. «Рисование» символами.

а) выведите на экран числа в следующем виде:

```
1
2       2
3       3       3
4       4       4       4
5       5       5       5       5
```

и т.д.

б) выведите на экран числа в следующем виде:

```
7       6       5       4       3       2
6       5       4       3       2
5       4       3       2
4       3       2
3       2
2
```

в) выведите на экран числа в следующем виде:

```
1
3
2       2
4       4
3       3       3
5       5       5
4       4       4       4
6       6       6       6
```

г) выведите звездочки «полуелочкой» (рис. 5.1) заданное количество раз.

```
*
* *
* * *
*
* *
* * *
* * * *
*
* *
* * *
* * * *
* * * * *
```

и т.д.

Рис. 5.1

д) превратите «полуелочку» в полную елочку.

6. Задачи на перебор вариантов

6.1. Перебор вариантов: теория

Имеется целый класс задач, решение которых сводится к перебору различных вариантов, среди которых выбирается такой, который удовлетворяет условию задачи.

Пример 1. Поиск делителей целого числа *N*.

Целое число *K* является делителем *N*, если остаток от деления *N* на *K* равен 0. Чтобы найти все делители, достаточно перебрать все числа от 1 до *N* и проверить, являются ли они делителями. Данный алгоритм можно реализовать с помощью программы:

```
readln(n);
for i:=1 to n do
   if n mod i = 0 then
      write(i, '   ');
```

На этом простейшем примере ясна общая структура решения задач на перебор вариантов. Для организации перебора используется цикл, каждый шаг выполнения которого соответствует рассмотрению одного из вариантов. Внутри цикла стоит проверка, подходит ли данный вариант под условие задачи.

Решая задачи методом перебора, всегда следует подумать, а нельзя ли каким-то образом сократить количество перебираемых вариантов. В данном случае заметим, что любое число делится само на себя и на 1. Поэтому эти варианты можно исключить из перебора. Более того, наибольшим делителем, отличным от самого числа *N*, может быть только *N*/2, а все числа, большие *N*/2, заведомо делителями не являются. Учет этих особенностей приводит к более эффективной программе:

```
readln(n);
write(1, '   ');
for i:=2 to n div 2 do
   if i mod i = 0 then
      write(i, '   ');
write(n);
```

Конец примера.

В приведенном примере вариантом решения являлась переменная i, используемая как счетчик цикла. Не трудно представить себе задачу, где такое невозможно. Например, вариант может не быть целым числом. В этом случае переменная-счетчик цикла выступает как номер варианта, а в цикле требуется совершить еще одно действие: по номеру определить этот самый вариант. Иными словами, вместо схемы: *перебираем варианты и проверяем, удовлетворяют ли они условию* будем иметь схему: *перебираем номера вариантов, по номеру вычисляем вариант решения, проверяем.*

Пример 2. Найти минимумы функции $f(x) = x^4 - x^2$ с точностью до 0.001 на отрезке от –5 до 5.

Для поиска минимума будем перебирать все числа от –5 до 5 с шагом 0.001. Условием нахождения минимума будем считать то, что значение функции $f(x_{\min})$ меньше, чем в соседних точках. А именно:

$$f(x_{\min}) < f(x_{\min} - 0.001)$$

и

$$f(x_{\min}) < f(x_{\min} + 0.001).$$

Данный алгоритм можно реализовать с помощью программы:

```
Step:=0.001;
MinX:=-5;
MaxX:=5;
{Вычисляем количество перебираемых вариантов решения}
VarNumber:=trunc((MaxX - MinX)/Step)+1;
for i:=1 to VarNumber do
begin
   {Вычисляем вариант, соответствующий номеру i}
   x:=MinX + (i-1)*step;
   {Проверяем вычисленный вариант}
   if (sqr(sqr(x)) -sqr(x) < sqr(sqr(x-step)) -
sqr(x- step)) and ((sqr(sqr(x)) -sqr(x) <
sqr(sqr(x+step)))
   then
      writeln(x);
end;
```

Кроме перебора вариантов, в начале программы использован еще один важный прием, называемый *параметризацией*. Такие характеристики, как точность и диапазон поиска были записаны в отдельные переменные, а затем вместо чисел использовались имена этих переменных. Такой подход позволяет легче модифицировать программу, если параметры задачи изменятся, программа становится более универсальной, а также понятнее, особенно если имена переменных выбраны так, чтобы соответствовать смыслу хранимого в них параметра (MinX – минимальное значение переменной x и т.п.)

Каждый следующий проверяемый вариант можно также вычислять не по номеру

```
x:=MinX + (i-1)*step;
```

а по предыдущему значению

```
x:=x + step;
```

Очевидно, это ничем не хуже.

Пример 3. Пусть двумя числами (H и V) задано положение коня на шахматной доске. Найдите координаты всех клеток, куда конь может пойти следующим ходом (других фигур на доске нет).

В данном примере вариантами решения являются координаты клеток, т. е. каждый вариант – это не одно число, а два. Таким образом, вновь необходимо придумать способ вычисления варианта (в данном случае номеров горизонтали и вертикали) по его номеру.

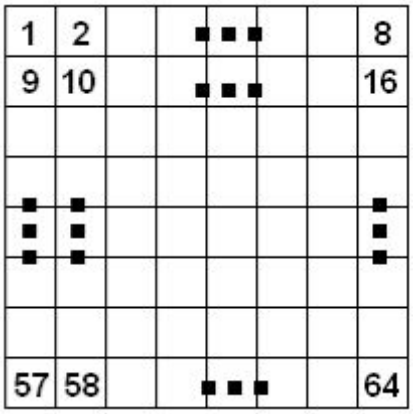


Рис. 6.1

Всего клеток 64. Пронумеруем клетки, как показано на рис. 6.1. Теперь, если мы будем перебирать в цикле номера клеток *N*, необходимо уметь по этим номерам определять номер горизонтали *X* и вертикали *Y*. Нетрудно видеть, что номер горизонтали определяется тем, сколько раз по 8 клеток надо взять, пока мы не доберемся до числа *N*:

```
X := N div 8 + 1;
```

Остаток после удаления целого числа раз по 8 клеток позволит вычислить номер вертикали:

```
Y := N mod 8;
```

Теперь, когда мы умеем перебирать варианты, необходимо записать условие того, что данный вариант является решением задачи. Известно, что конь ходит буквой Г, смещаясь на две клетки в одном направлении и на одну в другом. Например, на рис. 6.2 показан ход, когда перемещение коня состоит из сдвига на две клетки по вертикали и на одну по горизонтали.

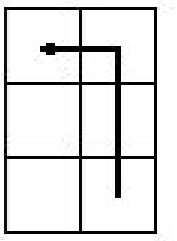


Рис. 6.2

Пусть проверяется клетка с координатами (*x*, *y*). Тогда условие сдвига по вертикали на 2 будет выглядеть так:

abs(x–H) = 2.

Взятие модуля необходимо, чтобы учесть возможность сдвига как вправо, так и влево. Аналогично условие сдвига на одну клетку, например, по вертикали:

abs(y–V) = 1.

Полное условие возможности пойти на клетку конем будет выглядеть как

```
((abs(x-H) = 2)and(abs(y-V) = 1)) or
                    ((abs(y-V) = 2)and(abs(x-H) = 1)).
```

Иными словами, либо сначала по горизонтали на две клетки, а потом на одну по вертикали, или на две по вертикали, потом на одну по горизонтали.

Теперь, когда ясно, как перебирать клетки и как проверять возможность хода, можно написать требуемую программу:

```
readln(H, V);  {запрашиваем расположение коня}
for n:=1 to 64 do
begin
   x := n div 8 + 1;
   y := n mod 8;
   if ((abs(x-H) = 2) and (abs(y-V) = 1)) or ((abs(y-
V) =
   2) and (abs(x-H) = 1)) then
      writeln(x, y);
end;
```

В случае когда как в приведенном примере каждый вариант задается не одним числом, а несколькими, удобно для перебора использовать вложенные циклы. В данном случае номера вертикали и горизонтали являются целым числами от 1 до 8, поэтому для перебора их значений можно использовать счетчики циклов:

```
readln(H, V);
for x:=1 to 8 do
   for y:=1 to 8 do
      if  ((abs(x-H)  =  2)  and  (abs(y-V)  =  1))  or
((abs(y-V)
      = 2)and(abs(x-H) = 1)) then
         writeln(x, y);
```

Рассмотрим еще один пример, где вариант задается несколькими числами.

Пример 4. Составить программу-генератор пифагоровых троек. Пифагоровой тройкой называют такие целые числа *a*, *b* и *c*, которые удовлетворяют условию $a^2 + b^2 = c^2$. Такие числа могут быть сторонами прямоугольного треугольника. Найти такие числа для $c \le 25$.

Тройное вложение циклов позволит перебрать все возможные комбинации значений трех чисел *a, b* и *c* и вывести те из них, которые удовлетворяют заданному равенству:

```
MaxC := 25;        {снова используем параметризацию}
MaxAB:=trunc(sqrt(MaxC));
for a:=1 to MaxAB do
   for b:=1 to MaxAB do
      for c:=1 to MaxC do
         if a*a+b*b = c*c then
         begin
            write(a, '   ', b, '   ', c);
            writeln;
         end;
```

Как всегда при решении задачи методом перебора, следует задуматься, можем ли мы сократить число перебираемых вариантов. Оказывается, можно ограничиться только перебором *a* и *b*, а *c* вычислять по теореме Пифагора: $c = \sqrt{a^2 + b^2}$. Вычисленное таким образом *c* может оказаться нецелым, поэтому условие задачи для него проверять все равно необходимо:

```
MaxC := 25;  {используем параметризацию}
MaxAB := trunc(sqrt(MaxC));
for a := 1 to MaxAB do
begin
   for b := 1 to MaxAB do
   begin
      c := round(sqrt(a * a + b * b));
      if a*a+b*b = c*c then
      begin
         write(a, '   ', b, '   ', c);
         writeln;
      end;
   end;
end;
```

Задание 6. Задачи на перебор вариантов

1. Для заданного целого числа проверьте, является ли оно кубом другого целого числа.

2. Задача Ал-Хорезми. Разложить число 10 на 2 слагаемых, сумма квадратов которых равна 58.

3. Задача Л. Эйлера. Некий чиновник купил лошадей и быков на сумму 1770 талеров. За каждую лошадь он уплатил по 31 талеру, а за каждого быка по 21 талеру. Сколько лошадей и быков купил чиновник? Используйте прием параметризации, чтобы легче было модифицировать программу при изменении рыночных цен и финансовых возможностей чиновника.

4. Теорема Ферма утверждает, что не существует решения в целых числах уравнения $x^n + y^n = z^n$ при $n > 2$. Напишите программу, проверяющую это утверждение при заданном *n* для всех *x*, *y* и *z* меньших 100.

5. Найдите все трехзначные числа, сумма цифр которых равна произведению цифр.

6. Решите следующие числовые ребусы:

УДАР + УДАР = ДРАКА

БУЛОК + БЫЛО = МНОГО

КОКА + КОЛА = ВОДА

ПОДАЙ – ВОДЫ = ПАША

ТРИ + ТРИ + ТРИ = ДЫРА, причем (Ы + Ы) / Ы = Ы

Здесь каждая буква взаимно-однозначно соответствует какой-нибудь цифре. Первая цифра числа не может быть нулем.

7. Переменные-флаги

7.1. Переменные-флаги: теория

Флаг – это полотнище правильной (как правило, прямоугольной) формы, прикрепленное к древку или поднимаемое на специальной мачте (флагштоке). Исторически флаги появились для передачи простых сигналов на поле боя. Например: подняли флаг, и конница понеслась в атаку! Как-то так. В простейшем случае с помощью флага передается информация объемом 1 бит (одно из двух: флаг поднят или нет).

Переменная-флаг – это, как правило, переменная логического типа, значение которой сигнализирует о состоянии вычислительного процесса. В каких случаях результат может характеризоваться одной только логической переменной? Приведем несколько примеров:

1) подводится баланс коммерческого предприятия. Дальнейшие действия могут зависеть от того, будет он положительным или отрицательным. Если отрицательный, надо просить кредит, положительный – планировать отдых на Багамах. В общем, самая существенная информация может быть передана одним битом;

2) решаем квадратное уравнение. Если дискриминант не отрицательный – ищем корни. Для хода вычислительного процесса важен факт неотрицательности, который также содержит 1 бит информации и может, таким образом, быть сохранен с помощью логической переменной;

3) детям на уроке физкультуры велено построиться по росту. Если они построились не по росту, надо на них наорать. Опять действия учителя зависят от информации объемом 1 бит.

Но кто и кому может передавать информацию в ходе выполнения программы? Дело в том, что при разработке больших программ происходит разделение задачи на более мелкие подзадачи (блоки), каждая из которых решается отдельно и, может быть, даже разными людьми. В этом случае один блок, закончив свою работу, должен передать ее результат другому блоку. Здесь и могут пригодиться флаги.

Пример 1. Решение квадратного уравнения.

Предположим, что программу решения квадратного уравнения пишут два человека (ну, можно же).

Пусть первый умеет решать квадратные уравнения, но не знает, как вывести результат (не ходил на лекции).

Второй знает, как вывести результат, но про квадратные уравнения слышит первый раз в жизни. Попробуем представить, какую программу они напишут:

```
var
   a, b, c: real;        {коэффициенты          уравнения
                                 ax^2 + bx + x = 0 }
   d: real;              {дискриминант}
   x1, x2: real;         {корни}
   RootsExist: boolean; {переменая-флаг}
begin
   {Блок, написанный первым программистом}
   readln(a, b, c);
   d := sqr(b)-4*a*c;
   RootsExist := (d>=0);{флаг служит для хранения ин-
                         формации о наличии корней}
   if RootsExist then
   begin
      x1 := (-b+sqrt(d))/(2*a);
      x2 := (-b-sqrt(d))/(2*a);
   end;
   {Блок, написанный вторым программистом}
   if RootsExist then
      writeln(x1, '   ', x2)
   else
      writeln('Roots don't exist');
end.
```

Переменная-флаг передает информацию о наличии корней. С ее помощью первый блок сигнализирует второму блоку. Вы скажете, почему бы второму программисту вместо строчки

```
if RootsExist then
```

не написать

```
if d >= 0 then
```

Зачем использовать еще одну переменную? Давайте вспомним, что второй программист ничего не знает о квадратных уравнениях и, в частности, не в курсе, что наличие корней определяется знаком дискриминанта. Не вдаваясь в тонкости чужой задачи, он просто просит первого программиста передать существенную информацию через переменную-флаг.

На таком простом примере, как решение квадратного уравнения, трудно проникнуться ощущением полезности флагов. Пожалуй, согласимся, что в данном случае это лишнее. Однако нетрудно представить себе более сложную задачу, где такой подход окажется полезным.

Представьте себе, что вам требуется написать программу размером несколько тысяч строк (это сравнительно небольшая программа). Есть единственный способ создавать программы такого размера – разбиение решаемой задачи на подзадачи и написание отдельных блоков программы,

решающих каждый свою подзадачу. Чтобы можно было сосредоточиться на решении отдельной подзадачи, надо сделать их решение по возможности максимально независимым друг от друга. Для этого от одного блока к другому должно передаваться как можно меньше информации. Так что даже если программу пишет всего один человек, флаги облегчат его работу.

Как уже отмечалось, флаги минимизируют информацию, передаваемую между блоками. Так, в примере с квадратным уравнением использование флага позволило передавать всего 1 бит вместо 6-ти байт, которые пришлось бы потратить на значение дискриминанта. Общий принцип здесь такой – чем меньше информации, тем труднее допустить ошибку. При разработке сложных программ поиск ошибок занимает больше времени, чем собственно их написание, и любая возможность уменьшить вероятность их появления должна приветствоваться.

В дополнение еще один пример.

Пример 2. Проверка упорядоченности последовательности.

Пользователь вводит 10 чисел. Требуется проверить, упорядочены ли они по возрастанию, и передать эту информацию с помощью переменной флага.

Решение:

```
Growing := true;        {переменная-флаг}
readln(x);

for i:=2 to 10 do
begin
   x2 := x;
   readln(x);
   Growing := Growing and (x<x2);
end;
```

Если очередное введенное число (x) будет больше предыдущего (x2), то флаг примет значение false и сохранит его до конца цикла.

Блоки, передающие друг другу информацию с помощью флагов, не обязательно должны идти последовательно друг за другом. Можно представить себе ситуацию, когда один блок является составной частью другого.

Пример 3. Найти все простые числа от 1 до *N*.

Число называется простым, если не делится ни на какое другое число, кроме 1 и самого себя. Простейший алгоритм поиска таких чисел состоит в том, чтобы перебрать все числа и для каждого проверить наличие делителей. Поиск делителей можно мыслить себе как отдельный блок программы, результатом работы которого будет присваивание значения флаговой переменной. Блок поиска простых чисел будет включать в себя блок

проверки на наличие делителей. Программную реализацию этого алгоритма выполните в качестве самостоятельного упражнения.

Не обязательно использовать в качестве флага именно логическую переменную. В принципе флагом может считаться любая переменная, принимающая небольшое количество возможных значений, каждое из которых характеризует тот или иной результат вычислительного процесса.

В примере с квадратным уравнением можно было бы предусмотреть еще одну ситуацию, когда $a = 0$, т. е. уравнение не квадратное. Тогда для передачи информации в следующий блок можно использовать либо две переменные логического типа, либо одну, но принимающую три значения (в качестве таковой можно использовать, например, переменную целого типа).

Задание 7. Переменные-флаги

1. Найдите все простые числа от 1 до 100. Используйте параметризацию, чтобы потом легко было искать простые числа в других диапазонах. Составной частью программы должен быть блок, проверяющий наличие делителей и записывающий эту информацию в переменную-флаг.

2. Пусть имеется корабль с 10 грузовыми отсеками. Максимальная грузоподъемность корабля 100 тонн. Пока не заполнены все отсеки или не достигнута максимальная грузоподъемность, ваша программа должна запрашивать массу груза, помещаемого в очередной отсек. Как только дальнейшая загрузка станет невозможной, программа должна прекратить запрашивать массу новых грузов и вывести сообщение о причинах этого прекращения: «Трюм полон» или «Достигнута максимальная грузоподъемность». Запрос массы грузов и вывод результатов загрузки рассматривайте как отдельные блоки программы, информация между которыми передается с помощью переменной-флага.

3. Последовательность чисел, заданная формулой $y_n = \sin x^n$, не содержит отрицательных элементов при малых значениях x. С точностью до 0.0001 определите, начиная с какого x отрицательные числа появляются среди первых 20 элементов этой последовательности. Используйте переменную-флаг для обозначения наличия или отсутствия отрицательных чисел в последовательности.

8. Переменная-счетчик событий

8.1. Переменные-счетчики

Часто требуется подсчитать, сколько раз во время вычислений наступает то или иное событие (выполняется то или иное условие). Для этого вводится вспомогательная переменная, которой в начале присваивается нулевое значение, а после каждого наступления события она увеличивается на единицу.

Пример 1. Пользователь вводит 10 чисел. Определить, сколько из них являются одновременно четными и положительными.

```
Counter := 0;                {обнуляем переменную-счетчик}
for i:=1 to 10 do
begin
   readln(x);
   if (x mod 2 = 0)and(x>0) then
      Counter := Counter + 1; {при выполнении условия
                               увеличиваем на 1}
end;
writeln(Counter);
```

Пример 2. Пользователь вводит 10 чисел. Проверить, упорядочены ли они по возрастанию.

Эту задачу мы решали в параграфе, посвященном переменным флагам, с помощью логической переменой. Решим ее теперь с помощью счетчика. Последовательность будет упорядочена, если нет ситуаций, когда последующее число меньше предыдущего. Подсчитаем количество таких ситуаций, и если оно окажется нулевым, то последовательность упорядочена:

```
Counter := 0;
readln(x);
for i := 2 to 10 do
begin
   x2 := x;
   readln(x);
   if x2 > x then Counter : =Counter + 1;
end;
if Counter = 0   then
   writeln('Последовательность упорядочена')
else
   writeln('Последовательность не упорядочена');
```

Пример 3. Вычисление площади сложных фигур *методом Монте-Карло*.

Метод Монте-Карло позволяет приближенно подсчитать площадь произвольной плоской фигуры. Для этого плоскую фигуру следует поместить внутрь простой, площадь которой легко подсчитать (например, внутрь квадрата). Затем следует случайным образом генерировать координаты точек так, чтобы они с равной вероятностью оказывались в любом месте квадрата. При этом следует подсчитать, какая доля случайных точек окажется внутри фигуры. При достаточно большом числе точек отношение количества точек, попавших внутрь фигуры, к общему количеству будет равно отношению площадей фигуры и квадрата.

Рассчитаем таким способом площадь под кривой графика $y = \sin x$ (см. рис. 8.1) в диапазоне от 0 до π.

```
N := 1000;   {Общее количество точек}
C := 0;  {Инициализируем счетчик}
for i := 1 to N do
begin
   {Генерируем слу-
    чайные координа-
    ты точки внутри
    прямоугольника}
   x := Pi * random;
   y := random;
   {В случае попадания в область под кривой увеличи-
    ваем счетчик}
   if y < sin(x) then C := C + 1;
end;
{Подсчитываем площадь фигуры}
S := Pi * C / N;
writeln(S);
```

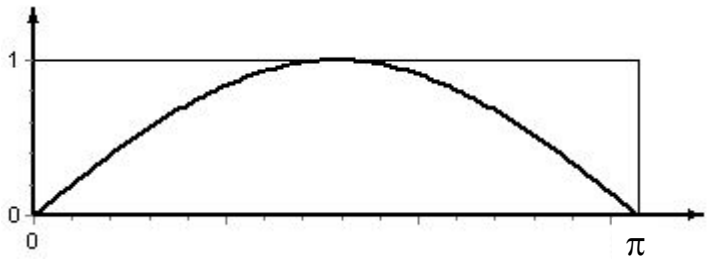


Рис. 8.1

Задание 8. Переменная-счетчик событий

1. Создайте программу, запрашивающую у пользователя 10 чисел. Если больше 4 из них окажутся больше 10, выведите сообщение «Караул! Сейчас все взорвется». Иначе сообщите, сколько введенных чисел больше 10, а сколько больше 5.

2. Напишите программу, которая генерирует *n* случайных чисел, способных принимать значения в диапазоне [–1; 2], и подсчитывает, сколько среди них отрицательных.

3. С помощью метода Монте-Карло получите приближенное значение числа π. Для этого подсчитайте площадь окружности единичного радиуса.

4. Напишите программу для подсчета числа точек с целочисленными координатами, находящихся внутри круга с центром в начале координат и радиусом 1000.

9. Циклы *while* и *repeat*

9.1. Синтаксис циклов *while* и *repeat*

В отличие от цикла **for** циклы **while** (оператор цикла с предусловием) и **repeat** (оператор цикла с постусловием) позволяют повторять выполнение тех или иных операторов не фиксированное количество раз, а только пока выполняется некоторое условие. Структура записи этих операторов выглядит следующим образом:

```
while <Условие> do
begin
  <Операторы>
end;
```

<Условие> – любое логическое выражение.
<Операторы> – любые операторы.

Перед каждым выполнением тела цикла анализируется значение выражения <Условие>. Если оно истинно (true), выполняется тело цикла. Затем снова проверяется условие и т.д. Если значение условия ложно (false), то работа цикла завершается. Если результат условия окажется ложным при первой проверке, то тело цикла не выполнится ни разу.

Цикл **repeat** работает похожим образом, однако в нем условие проверяется в конце цикла (после выполнения каждого шага). Структура записи оператора выглядит следующим образом:

```
repeat
   <Операторы>
until <Условие>;
```

Один раз тело цикла будет выполнено в любом случае. Затем будет проверено условие, и если оно истинно, то выполнение цикла повторится. Повторение продолжается пока не выполнится условие, стоящее после слова **until** («пока не»). Таким образом, если в цикле **while** мы задаем условие для продолжения повторений, то в случае **repeat**‘а ставится условие на прекращение повторений. Блок-схемы работы циклов **while** (*а*) и **repeat** (*б*) представлены на рис. 9.1.

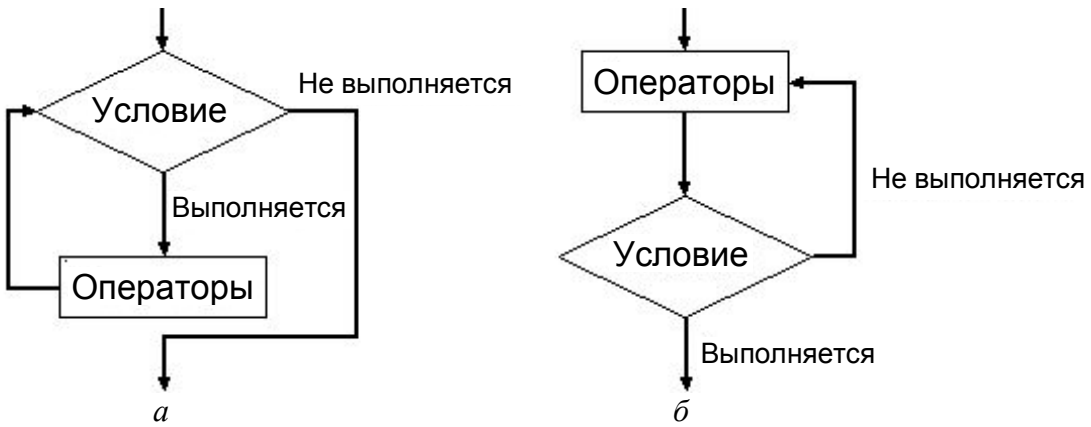


Рис. 9.1

Пример 1. Имитация работы цикла *for*.

Циклы с условиями более универсальны, чем **for**. С их помощью можно легко сделать все то же самое, только об изменении значения переменной-счетчика придется позаботиться самостоятельно. В качестве примера напечатаем числа от 1 до 10:

```
i := 1;
while i <= 10 do
begin
   writeln(i);
   i := i + 1;
end;
```

```
i := 1;
repeat
   writeln(i);
   i := i + 1;
until i > 10;
```

Заметим, что в отличие от цикла **for**, где увеличение счетчика всегда происходит на единицу, здесь мы сами управляем счетчиком и можем организовать его увеличение на любую требуемую величину. Так, в следующем примере используется увеличение счетчика на 2.

Пример 2. Напечатайте все нечетные числа от 3 до 25.

```
i := 3;
while i <= 25 do
begin
   writeln(i);
   i := i + 2;
end;
```

В большинстве случаев циклы **while** и **repeat** взаимозаменяемы. Цикл **repeat** предпочтительнее, если известно, что тело цикла необходимо выполнить хотя бы один раз.

9.2. Зацикливание

Если логическое выражение в цикле **while** будет всегда истинным, то работа такого цикла не завершится никогда. Такая ситуация называется *зацикливанием*.

Простейший способ создать такую ситуацию:

```
while true do
begin
   writeln('У попа была собака, он ее любил.');
   writeln('Она съела кусок мяса - он ее убил.');
   writeln('Вырыл ямку, закопал и на камне напи-
сал:');
end;
```

Аналогично можно задать конструкцию с циклом **repeat**, только условие будет условием выхода из цикла и соответственно должно быть неистинно.

Чаще, однако, такие ситуации возникают по ошибке. Для примера рассмотрим печать чисел от 1 до 10:

```
i := 1;
repeat
   writeln(i);
```

```
   //i := i + 1; - Представьте, что вы забыли напи-
сать эту строку
until i > 10;
```

Если вы забудете строку с увеличением счетчика, то i никогда не станет больше 10. Это настолько распространенная ошибка, что рекомендуется первым делом писать увеличение счетчика, а только потом возвращаться назад и писать все остальные операторы в теле цикла.

9.3. Цикл, управляемый меткой

Циклом, управляемым меткой, называется такой цикл, в теле которого на каждом шаге происходит запрос данных у пользователя, а сигналом к выходу из цикла служит ввод пользователем так называемой «метки выхода».

Для примера создадим программу, которая запрашивает у пользователя числа и подсчитывает их сумму. Количество чисел заранее не оговаривается, меткой выхода служит ввод числа 0:

```
s := 0;
repeat
   readln(x);
   s := s + x;
until x = 0;
writeln(s);
```

В данном случае выгоднее использовать **repeat**, а не **while**, так как хотя бы один запрос числа придется сделать. В случае **while**‘а этот первый запрос пришлось бы делать до цикла:

```
s := 0;
readln(x); {Запрос первого числа}
while x <> 0 do
begin
   s := s + x;
   readln(x);
end;
writeln(s);
```

9.4. Вычисление номера шага

Когда в параграфе 9.1 с помощью циклов с условиями имитировалась работа цикла **for**, счетчик шагов использовался для определения момента выхода из цикла. Однако легко представить себе ситуацию, когда выход из цикла осуществляется по какому-нибудь другому условию, а счетчик служит для определения числа шагов, потребовавшегося для вычислений.

Пример 3. Коммерсант, имея стартовый капитал *k* рублей, занялся торговлей, которая ежемесячно увеличивает его капитал на *p* процентов. Через

сколько лет он накопит сумму *s*, достаточную для покупки собственного магазина?

Решение:

```
x := k;                          {начальная сумма  равна k}
n := 0;                          {обнуляем счетчик шагов}
while x < s do                   {пока сумма не достигнет s}
begin
   x := x * (1 + p / 100);       {увеличиваем ее на p про-
                                  центов}
   n := n + 1;                   {увеличиваем счетчик шагов
                                  на 1}
end;
writeln(n  div  12,  '  years  and  ',  n  mod  12,  '
                                  months');
```

9.5. Вычисления с заданной точностью

При реализации многих численных методов точность вычислений зависит от числа шагов. Однако за какое именно число шагов будет достигнута приемлемая точность, заранее сказать трудно, и желательно, чтобы программа сама определяла, когда следует остановиться.

Например, синус можно разложить в так называемый ряд Тейлора:

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Чем большее количество членов ряда будет просуммировано, тем точнее будет вычислен синус. Пусть требуется вычислить до 5-го знака после запятой. Иными словами, приемлемая погрешность $\varepsilon = 10^{-5}$. Для этого достаточно суммировать члены ряда до тех пор, пока очередной его член не окажется меньше 10^{-5}:

```
eps := 1e-5;
readln(x);
p := x;
s := x;
n := 2;
while p <= eps do {условие выхода: очередной член ря-
                   да меньше eps}
begin
   p := -p*x*x/(n*(n+1)); {вычисление очередного чле-
                           на ряда}
   s := s+p;
   n := n+2;
end;
writeln('sin(x) = ', s);
```

Если вычисления производятся в соответствии с рекуррентными соотношениями, то еще один способ поставить связанное с точностью условие прекращения вычислений заключается в следующем. Вычисления прекращаются, если изменение вычисляемой величины на очередном шаге меньше заданной величины:

$$\left|x_{n+1} - x_n\right| < \varepsilon.$$

Также условие можно наложить на относительное изменение:

$$\left|\frac{x_{n+1} - x_n}{x_n}\right| < \varepsilon.$$

Контрольная работа 7

1. Сколько раз выполнится цикл?

а)
```
x := 1;
while x > 0.1
do
   x := x/2;
```

б)
```
x := 1;
y := 1;
while (x>10)or(y<-1)
do
begin
   x := x*2;
   y := y-1;
end;
```

в)
```
i := 1;
x := 0;
while i < 5 do
   x := x*2;
```

2. Чему равны переменные *x* и *y* после выполнения операторов?

а)
```
x := 1;
y := 0;
while x < 10 do
begin
   x := x + y;
   y := y + 1;
end;
```

б)
```
x := 1;
y := 2;
for i := 1 to 3 do
   while x < i*3 do
   begin
      x := x + y;
      y := y + 1;
   end;
```

3. Чему равны переменные A и B после выполнения операторов?

```
A := 45;
B := 18;
while A <> B do
   if A > B then
     A := A - B
   else
     B := B - A;
```

4. При выполнении следующей программы пользователь ввел числа 1, 20, 17, 6, 10, 13. Какое число выведет программа?

```
readln(x);
m := x;
while x <> 13 do
begin
   readln(x);
   if x > m then
     m := x;
end;
writeln(m);
```

Задание 9. Циклы *while* и *repeat*

1. Напечатайте на экране 10 раз слово Hello с помощью цикла **while** и столько же с помощью цикла **repeat**.

2. Напечатайте в столбик нечетные числа от 3 до 25.

3. Найдите минимальное число большее 300, которое делится на 19.

4. Определите, является ли введенное пользователем число степенью тройки (не используя логарифмы).

5. Из математического анализа известно, что последовательность сумм вида

$$s_n = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + (-1)^n \frac{x^{2n}}{(2n)!}$$

сходится к функции $\cos x$. Напишите программу, которая будет суммировать этот ряд до тех пор, пока очередная добавка не окажется меньше 10^{-6}.

6. Напишите программу, которая запрашивает у пользователя числа до тех пор, пока каждое следующее число больше предыдущего. В конце программа сообщает, сколько чисел было введено.

7. Последовательность Фибоначчи определяется рекуррентным соотношением $x_{n+1} = x_n + x_{n-1}$, где $x_0 = 1$, $x_1 = 1$. Найдите первое число в последовательности Фибоначчи, которое больше 1000.

8. Для *n*-го члена в последовательности Фибоначчи существует явная формула $x_n = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right)$. Поскольку операции с вещественными числами происходят с конечной точностью, то с ростом *n* результат вычисления по этой формуле будет все больше отличаться от настоящего числа Фибоначчи. Найдите *n*, начиная с которого, отличие от истинного значения составит 0.001.

9. Создайте программу, играющую с пользователем в орлянку. Программа должна спрашивать у пользователя, орел или решка? Если пользователь вводит 0, то выбирает орла, 1 – решку, любое другое число – конец игры. Программа должна вести учет выигрышей и проигрышей и после каждого раунда сообщать пользователю о состоянии его счета. Пусть вна-

чале на счету будет 1 рубль и ставка в каждом коне тоже 1 рубль. Если денег у пользователя не осталось, игра прекращается.

10. Усовершенствуйте разработанный в предыдущем задании «игровой автомат» таким образом, чтобы выигрыш происходил только в 40% случаев.

11. В 1593 году Франсуа Виет предложил для вычисления числа π формулу

$$\frac{2}{\pi}=\sqrt{\frac{1}{2}}\cdot\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}}}\cdot\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}+\frac{1}{2}\sqrt{\frac{1}{2}}}}\cdot\ldots$$

В 1655 году профессор Оксфордского университета Джон Уоллис (John Wallis) предложил формулу

$$\frac{2}{\pi}=\frac{1\cdot 3}{2\cdot 2}\cdot\frac{3\cdot 5}{4\cdot 4}\cdot\frac{5\cdot 7}{6\cdot 6}\cdot\frac{7\cdot 9}{8\cdot 8}\cdot\ldots$$

Сообщив о ней лорду Брункеру (Lord Brouncker), он получил в ответ разложение

$$\frac{4}{\pi}=1+\cfrac{1}{2+\cfrac{9}{2+\cfrac{25}{2+\cfrac{49}{2+\ldots}}}}.$$

Наконец, 1674 году Г. Лейбниц показал, что число

$$\frac{\pi}{4}=1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\ldots$$

Определите, какой из этих способов обеспечивает более быструю сходимость.

10. Массивы

10.1. Структурные типы данных

Для большинства рассмотренных ранее типов данных было характерно два свойства: неделимость и упорядоченность их значений. Например, целое число есть объект, не распадающийся на компоненты (неделимость), а множество целых чисел упорядочено. Можно, конечно, сказать, что число состоит из цифр, но отдельная цифра, как правило, не имеет самостоятельного смысла. Скажем, есть число – количество студентов в группе. Что означает вторая цифра этого числа? По-видимому, ничего. Такие типы называются скалярными.

При обработке данных бывает важно иметь возможность дать коллективное имя целому множеству объектов. Типы, значения которых состоят из нескольких компонент, называют *структурными*.

Приведем примеры.

1. Оценка за контрольную конкретного студента – скалярное значение, а результаты контрольной по студенческой группе образуют структурное значение.

2. Вектор задается своими координатами. Отдельные координаты – скалярные значения, вектор целиком – структурное.

В обоих приведенных примерах каждое структурное значение состоит из набора скалярных значений одного типа. В этом случае используется один из самых простых структурных типов – *массивы*.

10.2. Основные определения

Переменная, имеющая структуру массива, является совокупностью компонент одного и того же типа. В математике наиболее близкими понятиями являются вектора и числовые последовательности, где целая группа чисел обозначается одним именем, а для обращения к каждому отдельному числу последовательности используются различные *индексы*. Выглядеть это может, например, так: $a_1, a_2, a_3, \ldots a_n$. Иными словами, обращаемся к конкретному элементу, например a_3, указав имя последовательности или вектора (a) и номер (индекс) элемента (3).

Для использования подобных объектов в программировании требуется сначала описать соответствующий тип в разделе описания типов. Это делается следующим образом:

```
type
   <имя типа-массива> = array [<тип индексов>] of <тип элементов>;
```

Конкретный пример:

```
type
   TMassive = array [1..10] of real;
```

Здесь <имя типа-массива> – произвольный правильный идентификатор, любое имя, которое вы захотите дать своему типу. Согласно правилам хорошего стиля имя начинается с большой буквы T (это позволяет отличать идентификаторы типов от всех прочих). Остальная часть имени придумывается в соответствии со смыслом данных, которые будут храниться в переменных этого типа, и пишется с заглавной буквы (опять это не обязательное требование, но желательное с точки зрения стиля). Примеры стилистически правильных имен:

TGrades – для хранения оценок за контрольную;

TVector – для хранения координат абстрактного вектора;

TMassiv – просто массив, когда не приходится задумываться, что означают его отдельные элементы (просто куча чисел);

<тип элементов> – произвольный стандартный или ранее введенный вами тип. Если это, например, тип real, то массив представляет собой несколько вещественных чисел;

<тип индексов> – в качестве индексов обычно используют целые числа, а в качестве типа индексов указывают так называемый тип-диапазон. Например, тип 1..10 означает, что массив будет состоять из 10 элементов, каждому из которых соответствует индекс – целое число от 1 до 10. Нумерация элементов не обязательно должна начинаться с единицы. Например, в качестве типа можно указать –5..5. В этом случае массив будет содержать 11 элементов, первому из которых соответствует индекс –5, второму –4 и т. д.

Приведем несколько примеров описания типов массивов:

`TMassive2 = array [0..100] of real;` – набор из 101 вещественного числа с индексами в диапазоне от 0 до 100.

`TNames = array [1..20] of string;` – массив, подходящий для хранения имен 20 человек.

`TLargeMassive = array [integer] of real;` – массив из 4294967296 (2^{32}) вещественных чисел. Каждая переменная такого типа будет занимать в памяти $2^{32} \cdot 8$ байт $= 16$ Гб, что довольно много. Именно поэтому рекомендуется заранее подумать, сколько элементов вам реально понадобится, и использовать тип-диапазон с нужным количеством возможных значений.

`TIntMassive = array [1..20] of integer;` – массив, значением которого будет набор из 20 целых чисел.

После того как описан тип, можно вводить его переменные.

Например:

```
var
   x, y: TMassive;
```

Можно обойтись и без введения специального типа, описывая переменные как

```
a, b: array [1..10] of real;
```

Однако впоследствии (при изучении темы «Процедуры и функции») потребуется именно введение типа, так что лучше привыкать сразу и всегда пользоваться заранее описанным типом.

В описанных переменных хранятся сразу 10 вещественных значений. Индексами являются числа от 1 до 10. Для простоты индексы можно понимать как номера хранящихся в массиве чисел. Элемент массива `a` с индексом `i` обозначается как `a[i]`. Например, `a[1]` – элемент с индексом 1. Каждый элемент можно рассматривать как переменную типа real и соответственно с ней обращаться – присваивать ему значения, использовать в выражениях и т.д. Например, операторы

```
a[1]:=2.5;
readln(a[2]);
a[3]:=2*a[2]+a[1];
```

запишут в 1-й элемент число 2.5, во второй – то, что пользователь введет с клавиатуры, а третий элемент будет вычислен по первым двум.

10.3. Вычислимость индексов

До сих пор массив ничем не отличался от набора однотипных элементов. В конце концов, мы могли бы сделать объявление:

```
a1, a2, a3, a4, a5, a6, a7, a8, a9, a10: real;
```

и также пользоваться этими переменными. Массивы разве что сокращали длину описания (представьте, что таких переменных вам понадобится 1000 штук). Однако есть радикальное отличие, столь важное, что разговор о нем выносится нами в отдельный раздел. Дело в том, что индексы элементов массива в отличие от номеров переменных в сделанном выше объявлении можно вычислять. Иными словами, для указания элемента массива можно использовать не только числа (1, 2 и т. д.), но и произвольное выражение, тип значения которого совместим с типом индекса.

Например, если объявлены переменные

```
var
   i: integer;
   x: array [1..10] of real;
   y: array [1..10] of integer;
```

допустимы следующие обращения к элементам массива x:

`x[2*2]` – обращение к 4-му элементу массива;

`x[i]` – обращение к элементу массива, индекс которого хранится в переменной i;

`x[2*i]` – индексом является удвоенное значение переменной i;

`x[i+1]` – снова индекс – арифметическое выражение;

`x[random(10)+1]` – случайный элемент массива;

`x[y[i]]` – в качестве индекса берется число, хранящееся в i-м элементе другого массива. Если значение y[i] не попадает в диапазон 1..10, то такое обращение вызовет ошибку.

10.4. Примеры программ, работающих с массивами

Пример 1. Заполнение массива случайными числами.

```
const
   n = 10;     {количество  элементов  массива  будем
               хранить в константе}
type
   TMassive = array [0..n-1] of real; {тип-массив из n
                     элементов типа real}
var
   x: TMassive;
   i: integer;
```

```
begin
   for i:=0 to n-1 do
      x[i]:=random;   {заполнение элементов массива
                      случайными числами}
      for i:=0 to n-1 do
      writeln(x[i]);  {вывод элементов массива}
end.
```

Как говорилось ранее, важным приемом программирования является параметризация. В данном случае мы работаем с массивом из 10 элементов, но всюду в программе используем не число 10, а константу n = 10. Это облегчает возможную модификацию программы в случае, когда придется работать с другим количеством элементов в массиве: достаточно изменить значение константы. Другой стилистической особенностью программы является разделение задач заполнения и вывода элементов. При желании это можно было сделать в одном цикле, но с точки зрения стиля **разделение задач всегда хорошо**.

Обратите также внимание, что нумерация элементов в массиве начинается с нуля. Кто-то скажет, что начинать с единицы удобнее. Однако лучше привыкнуть к варианту с началом из нуля. Во-первых, существуют так называемые динамические массивы, где другой вариант нумерации невозможен, и чтобы избежать путаницы, лучше привыкнуть к одному варианту. Во-вторых, в некоторых задачах вычислять индексы, начинающиеся с нуля, проще.

Пример 2. Вычисление среднего арифметического элементов массива.

Вычисление состоит из двух этапов – подсчета суммы элементов и ее деления на общее число элементов:

```
const
   n = 10;
type
   TMassive = array [0..n-1] of real;
var
   x: TMassive;
   s: real;
   i: integer;
begin
   … {Присвоение значений элементам массива x}
   s:=0;
   for i:=0 to n-1 do
      s:=s+x[i];
   s:=s/n;
   writeln('Mean value = ', s);
end.
```

Смысл программы кажется прозрачным и не нуждается в пояснениях.

Пример 3. Поиск элемента в массиве.

Пусть заданы вещественнозначный массив x[1], x[2], …, x[n], а также число *y*. Определить, есть ли в массиве элемент x[k], такой что x[k] = y, и чему равен его индекс *k*:

```
const
   n = 10;
type
   TMassive = array [0..n-1] of integer;
var
   x: TMassive;
   y: real;
   k: integer;
begin
   … {Присвоение значений элементам массива x}
   readln(y);
   k:=0;
   while (k < n)and(x[k] <> y) do
      k:=k+1;
   if k > n then
      writeln('No such element')
   else
      writeln('Element has number: ', k);
end.
```

Цикл **while** может закончиться по двум причинам: когда найден нужный элемент (x[k] = y) или когда были проверены все элементы ($k >= n$). Понять, что послужило причиной окончания цикла, можно по значению переменной *k*.

Заметим, что если в массиве нет искомого элемента, то на последнем шаге цикла $k = n+1$. По этой причине принципиально, в каком порядке стоят условия продолжения цикла **while**. Если поменять их местами, сделав (x[k]<>y)**and**(k<=n), это приведет к ошибке – обращению к несуществующему элементу x[n+1] на последнем шаге.

Пример 4. Поиск максимального элемента в массиве.

Дан числовой массив. Требуется найти его самый большой элемент, вывести его значение и индекс.

В принципе достаточно найти только индекс элемента. Зная индекс, всегда можем узнать значение. Идея алгоритма состоит в том, чтобы, перебирая элементы массива, запоминать в отдельной переменной индекс самого большого. Тогда, беря очередной элемент, требуется сравнить его с самым большим из предыдущих и, если он больше, запомнить его индекс.

```
const
   n = 10;
```

```
type
   TMassive = array [0..n-1] of integer;
var
   x: TMassive;
   k, Kmax: integer; {Kmax – переменная для хранения
                      индекса наибольшего из просмот-
                      ренных элементов}
begin
   … {Присвоение значений элементам массива x}
   Kmax := 0;    {предположим, что первый элемент –
                      самый большой}
   for k:=1 to n-1 do
      if x[k]>x[Kmax] then
         Kmax:=k;     {если очередной элемент больше
                      самого большого из  предыдущих,
                      запоминаем его номер. На следую-
                      щем шаге цикла новый элемент бу-
                      дет сравниваться уже с ним}
   writeln('Max = ', x[Kmax]);
   writeln('Max element index = ', Kmax);
end.
```

10.5. Сортировка массивов

Одна из важнейших задач, связанных с массивами, – сортировка, т. е. расположение элементов массива по возрастанию или убыванию. Пусть есть массив, содержащий 4 элемента:

x[1] = 0.5, x[2] = 0.75, x[3] = 0.12, x[4] = 0.62.

Сортировка будет заключаться в преобразовании этого массива к виду

x[1] = 0.12, x[2] = 0.5, x[3] = 0.62, x[4] = 0.75.

Чаще всего сортировка применяется для последующего поиска. Представьте себе, что вы ищете слово в англо-русском словаре, где слова располагаются в случайном порядке. При наличии сотни тысяч слов задача кажется затруднительной. Однако сортировка по алфавиту радикально облегчает поиск.

Есть и масса других применений.

Существует несколько алгоритмов сортировки. В зависимости от особенностей решаемой задачи эффективней окажется тот или иной алгоритм. Тем не менее если время работы алгоритма не критично, то стоит выбрать тот, который наиболее прост в реализации. Нам таким представляется *метод выбора*.

Идея следующая. На первом шаге в массиве ищется самый маленький элемент и меняется местами с первым. На втором шаге ищется самый маленький, начиная со второго, и ставится на 2-е место (второй элемент,

естественно, не забываем: ставим его на место, где раньше был самый маленький). И так далее, на i-м шаге ищется самый маленький, начиная с i-го, и ставится на i-е место. Программную реализацию этого метода оставим для самостоятельной реализации.

10.6. Хороший стиль при решении задач на массивы

Большинство правил уже упоминалось выше в предыдущих параграфах. Теперь просто сведем их воедино:

1. Количество элементов массива следует задавать с помощью константы. Далее в программе всюду следует использовать эту константу, а не задавать количество элементов явно цифрами. Так, во всех приведенных в параграфе 10.4 примерах использовалась константа *n*, а не само число элементов (10).

2. Следует описывать тип-массив и использовать его переменные.

3. Следует стремиться разделять такие задачи, как:

а) присваивание значений элементам массива,

б) вычисление чего-либо по элементам массива или преобразование массива (решение содержательной задачи),

в) вывод результата.

Не помещайте все это в один цикл. Пусть правилом станет наличие как минимум трех циклов в программе, обрабатывающей массив: цикл при вводе элементов, цикл при обработке и цикл при выводе результата.

Контрольная работа 8

1. Пусть описан тип-массив:

```
type
   TMas = array [-n..n] of real;
```

Какой индекс будет иметь второй по счету, третий, предпоследний и третий с конца элементы этого массива? Каким по счету идет элемент с индексом 0?

2. Пусть имеется массив с шестью целочисленными элементами, равными x[0] = 2, x[1] = 8, x[2] = 10, x[3] = 3, x[4] = 2, x[5] = –6.

а) какие значения примут элементы массива после выполнения операторов?

```
i := 2;
c := x[1+i];
x[1+i] := x[2*i-2];
x[2*i-2] := x[2*i-1];
x[2*i-1] := c;
```

б) какое число выведет программа?

```
s1 := 0;
s2 := 0;
```

```
for i := 0 to 2 do
begin
   s1 := s1 + x[2*i+1];
   s2 := s2 + x[2*i];
end;
writeln(s2 - s1);
```

в) какое число выведет программа?

```
s := 0;
for i := 0 to 2 do
   s := s + x[i] - x[6-i];
writeln(s);
```

г) какие значения примут элементы массива после выполнения операторов?

```
n:=6;
for i:=0 to 2 do
begin
   c:=x[n-1];
   for k:=n-2 downto 0 do
      x[k+1]:=x[k];
   x[0]:=c;
end;
```

3. Пусть имеется массив с элементами: x[0] = 6, x[1] = 3, x[2] = 9, x[3] = 1. Какие значения примут элементы массива после выполнения операторов?

а)
```
repeat
   n := 0;
   for i := 0 to 2 do
      if x[i] > x[i+1]
      then
      begin
         c := x[i];
         x[i] := x[i+1];
         x[i+1] := c;
         n := n+1;
      end;
until n = 0;
```

б)
```
N:=4;
for i:=0 to N-2 do
begin
   m:=i;
   for k:=i+1 to N-1 do
      if x[k] > x[m]
      then
         m:=k;
   c:=x[i];
   x[i]:=x[m];
   x[m]:=c;
end;
```

в)
```
N:=4;
m1:=0;
m2:=0;
for i:=1 to N-1 do
begin
   if x[i]>x[m1]
```

г)
```
N:=4;
repeat
   p:=1;
   for i:=0 to N-1 do
   begin
      p:=p*x[i];
```

```
        then
           m1:=i;
        if x[i]<x[m2]
        then
           m2:=i;
     end;
     c:=x[m2];
     x[m2]:=x[m1];
     x[m1]:=c;
```

```
           x[i]:=x[i]-1;
        end;
     until p<0;
```

Контрольная работа 9

1. Пусть имеется массив с элементами x[0] = 2, x[1] = 7, x[2] = 5, x[3] = 2, x[4] = 3. Что выведут следующие программы?

а)
```
N:=5;
for i:=0 to N-1 do
begin
   y[i]:=0;
   for k:=0 to N-1 do
      if x[k] = x[i]
      then
         y[i]:=y[i]+1;
      if y[i]>1 then
         writeln(x[i]);
end;
```

б)
```
N:=5;
for i:=0 to N-1 do
begin
   y[i]:=0;
   for k:=i+1 to N-1
   do
      if x[k] = x[i]
      then
         y[i]:=y[i]+1;
   if y[i] = 0 then
      writeln(x[i]);
end;
```

в)
```
N:=5;
for i:=0 to N-1 do
   y[i]:=N-1-i;
for i:=0 to N-1 do
   writeln(x[y[i]]);
```

г)
```
N:=5;
for i:=1 to N do
   y[i-1]:=N-i;
for i:=0 to N-1 do
   writeln(y[y[i]]);
```

2. Пусть имеется массив с элементами x[0] = 2, x[1] = 7, x[2] = 5, x[3] = 2. Что выведет программа?

```
N:=4;
for i:=0 to N-1 do
   y[i]:=i;
repeat
   n:=0;
   for i:=0 to N-2 do
      if x[i]>x[i+1] then
      begin
         c:=x[i];
         x[i]:=x[i+1];
         x[i+1]:=c;
```

```
            d:=y[i];
            y[i]:=y[i+1];
            y[i+1]:=d;
            n:=n+1;
         end;

until n=0;
for i:=0 to N-1 do
   writeln(x[y[i]]);
```

Задание 10. Массивы

1. Заполнение массивов.

Опишите три массива с одинаковым количеством элементов, заданным константой. Значения элементов первого массива должны вводиться с клавиатуры, второго – быть равными номерам элементов, третьего – быть случайными целыми числами в диапазоне от 0 до 10. После заполнения выведите элементы каждого из массивов.

При решении следующих задач можно использовать любой из способов заполнения.

2. Пусть описаны константа и два типа-массива:

```
const
   m = 3;
type
   TMas1 = array [1..2*m+1] of real;
   TMas2 = array [-m..m] of real;
```

Создайте программу, которая значения, записанные в элементах массива 1-го типа, переносит в массив 2-го типа.

3. Создайте программу, проверяющую, есть ли в целочисленном массиве хотя бы один нечетный элемент.

4. В массиве, элементами которого являются целые числа, произведите следующие действия:

а) Если элементы меньше заданного числа, замените их этим числом.

б) Замените все элементы с индексами в диапазоне [a, b] нулями.

в) Поменяйте местами первый и последний элементы массива.

г) Совершите циклическую перестановку элементов массива (сдвиньте все элементы вправо, а последний поставьте на первое место).

д) Расположите элементы массива в обратном порядке.

е) Поменяйте местами элементы с четными и нечетными индексами (1-й со 2-м, 3-й с 4-м и т.д.).

ж) Сожмите массив, выбросив из него каждый второй элемент. Оставшуюся половину массива заполните нулями.

з) Вставьте дополнительный элемент в массив в заданное место. Чтобы освободить это место, все элементы, начиная с него, сдвиньте вправо. Последний элемент будет некуда девать – пренебрежем этим.

и) Определите индекс указанного пользователем элемента массива. Если такого элемента нет, сообщите об этом пользователю.

5. Пусть имеется массив, заполненный целыми числами. Получите на его основе новый массив следующими способами:

а) Получите массив из разностей соседних элементов исходного массива.

б) Получите массив из скользящих сумм *n* соседних элементов. Иными словами, каждый *i*-й элемент нового массива – это сумма *n* элементов старого, начиная с *i*-го. Новый массив, очевидно, будет содержать на *n*-1 элементов меньше. Пусть массивы заданы с одинаковой длиной, просто не выводите элементы нового массива, которые невозможно вычислить.

6. Получите сумму и среднее арифметическое всех элементов массива.

7. Получите корень из суммы квадратов всех элементов массива (модуль вектора).

8. Для двух массивов получите сумму попарных произведений их членов (скалярное произведение).

9. Для двух массивов получите евклидово расстояние между ними.

10. Найдите максимальный и минимальный элементы массива.

11. Отсортируйте элементы массива методом выбора.

11. Процедуры и функции

11.1. Простейшая процедура

Часто одну и ту же последовательность инструкций требуется повторить в нескольких местах программы. Чтобы не тратить время на копирование инструкций, в большинстве языков программирования предусмотрены средства для организации *подпрограмм*. Подпрограмма – последовательность инструкций, которой можно дать произвольное имя и использовать его в качестве сокращенной записи. Такую именованную последовательность инструкций будем называть также *процедурой*. Определение сокращенной записи называется описанием процедуры, а вызов ее из программы – вызовом процедуры или оператором процедуры.

Пример 1. Создадим программу с процедурой, печатающей на экране слово *Hello*:

```
program HelloProc;
//<Раздел описания переменных, типов и т.д.>
```

```
procedure P1; {заголовок процедуры. P – имя процеду-
ры}
begin {начало тела процедуры}
   writeln('Hello');
end; {конец тела процедуры}
begin {начало программы}
   P1; {вызов процедуры}
   //<Еще какие-то инструкции>
   P1; {еще один вызов процедуры}
end. {конец программы}
```

На этом примере мы видим следующее:

1) Описание процедуры располагается в разделе описаний программы (там же, где описываются переменные, типы, константы и т.п.). Традиционно описание процедур ставится в конец раздела (после того как описаны все переменные), но это не обязательно.

2) Простейшая процедура состоит из заголовка и тела процедуры. Заголовок состоит из слова **procedure** и имени процедуры. Имена процедур (как и имена переменных, типов и констант) могут быть любым сочетанием латинских букв, цифр и символа подчеркивания, начинающимся не с цифры.

3) Тело процедуры ограничено словами **begin** и **end**. После **end**'а ставится точка с запятой. В теле процедуры пишутся все инструкции, которые будут выполняться при ее вызове.

4) Вызов процедуры производится в разделе операторов программы. Для вызова достаточно написать имя процедуры. В приведенном примере вызов производится два раза. Соответственно дважды будет напечатано слово *Hello*.

11.2. Локальные переменные

Каждая процедура может иметь собственный раздел описания переменных (а также типов, констант и т.п.). Переменные, описанные в этом разделе, называются *локальными* и действуют только внутри процедуры.

Часто некоторые переменные используются только внутри некоторой последовательности инструкций и не имеют смысла за их пределами. В программе существенно проще разобраться, если такие инструкции будут оформлены как процедура, а переменные будут описаны как локальные.

Пример 2. Программа с процедурой, печатающей слово *Hello* 10 раз.

```
var
   i: integer;
procedure Hello10;
var {открытие раздела описаний локальных переменных}
   i: integer; {описание локальной переменной i}
```

```
begin
   for i:=1 to 10 do
      write('Hello_');
end;

begin
   for i:=1 to 5 do
      Hello10; {пять раз вызываем процедуру}
end.
```

Программа выведет 5 строк по 10 слов *Hello*.

Как видим, раздел описаний процедуры располагается сразу после заголовка и перед телом процедуры. В разных процедурах можно использовать локальные переменные с одинаковыми именами. Несмотря на одинаковость имен, это будут разные переменные, их значения будут храниться в разных областях памяти, а присвоение значений одним никак не повлияет на значения других. После завершения работы процедуры память, выделенная под локальные переменные, освобождается, все их значения пропадают.

Переменные, описанные в разделе описаний основной программы, будем называть *глобальными*. Глобальные переменные действуют как в теле программы, так и во всех процедурах. Если имена локальной и глобальной переменной, как в приведенном примере, совпадают, то это все равно разные переменные. Внутри процедуры при обращении к переменой i подразумевается, что это локальная переменная.

Представьте себе, что в рассмотренном примере локальная переменная i не описана. Синтаксической ошибки не возникнет, так как есть глобальная переменная, которая и будет использована как счетчик цикла внутри процедуры. Однако смысл программы изменится. Процедура изменит значение глобальной переменной i, которая используется как счетчик в другом цикле уже в основной программе. Последствия непредсказуемы. Скорее всего, программа выведет только одну строку со словами *Hello*. Но она может и зациклиться, выводя *Hello* до бесконечности.

Пример 3. Глобальная и локальная переменные.

```
var
   x: real; {описание глобальной переменной x}
procedure P2;
var
   x: real; {описание локальной переменной x}
begin
   x:=10;  {присваивание значения локальной перемен-
               ной}
   writeln(x); {вывод значения локальной переменной}
end;
```

```
begin
   x:=5;  {присваивание значения глобальной перемен-
               ной}
   P2; {вызов процедуры}
   writeln(x); {вывод значения глобальной переменной}
end.
```

Несмотря на то что при вызове процедуры P2 выполнится инструкция x:=10, глобальная переменная x останется равной 5.

11.3. Параметры процедур

Очень часто некоторые последовательности инструкций, встречающиеся в программе, не идентичны, но очень близки по форме. Особенно важна ситуация, когда различие между вхождениями инструкций можно устранить систематической заменой идентификаторов или выражений.

Пример. В описанном в пункте 10.5 алгоритме сортировки методом выбора требовалось искать минимальный элемент в несортированной части массива и ставить его на левый край несортированной части. Отличие в действиях на каждом шаге алгоритма сводилось к изменению одного параметра – номера индекса, с которого начиналась несортированная часть.

Такие инструкции можно записать в виде процедуры, работа которой зависит от одного или нескольких параметров. Параметры описываются в скобках после имени процедуры.

Пример 4. Программа, печатающая произведение двух чисел с помощью процедуры.

```
var
   x, y: integer;

procedure Mult(a, b: integer); {заголовок процедуры с
                                  двумя параметрами}
var
   c: integer;
begin
   c:=a*b; {значения параметров используются для вы-
                                  числения c}
   writeln(c);
end;

begin
   readln(x, y);
   Mult(x, y); {вызов процедуры с указанием значений
                                  параметров}
end.
```

Программа запросит у пользователя значения переменных x и y, затем вызовет процедуру Mult, указав при этом, что значения параметров a и b при данном вызове процедуры должны быть равны значениям переменных x и y. Процедура выведет произведение `x*y`.

Переменные a и b в заголовке процедуры называются формальными параметрами. То, что подставляется при вызове процедуры (в примере это переменные x и y), называется фактическими параметрами.

Формальный параметр внутри процедуры может использоваться как обычная локальная переменная. Иными словами, их можно использовать в выражениях и даже присваивать им новые значения. Отличие от просто локальных переменных состоит в том, что при вызове процедуры обязательно указывается, чему они изначально равны.

В качестве фактических параметров в данном примере могут выступать произвольные выражения. Например, возможен такой вызов процедуры: `Mult(x/y+1, 2)`. Значения этих выражений присвоятся формальным параметрам, и процедура напечатает значение выражения `(x/y+1)*2`.

11.4. Параметры-значения и параметры-переменные

Существует два способа описывать параметры: как *параметры-значения* и как *параметры-переменные*. В предыдущем параграфе использовались параметры-значения. Параметр-значение является просто локальной переменной, начальное значение которой задается при вызове процедуры.

Пример 5. Исползование глобальных переменных в процедуре.

```
var
   a, b: integer; {глобальные переменные}
procedure P3(a, b: integer); {внутри процедуры симво-
             лами a и b обозначаются формальные пара-
             метры. Действия с ними никак не повлияют
             на значения глобальных переменных a и b}
begin
   a:=a+1;
   b:=b+1;
   writeln(a+b);
end;
begin
   a:=1;
   b:=1;
   P3(a, b);  {вызов процедуры, где в качестве факти-
              ческих параметров использованы значения
              глобальных переменных a и b}
   writeln(a, b);
end.
```

Если в заголовке процедуры перед какими-либо из параметров поставить слово **var**, то это будут параметры-переменные. Например:

```
procedure P3(var a, b: integer; c: real);
```

Здесь a и b – параметры-переменные, c – параметр-значение. При вызове процедуры фактический параметр, задающий значения этих формальных параметров, не может быть выражением или константой, а должен быть обязательно переменной. Иными словами, недопустимы следующие вызовы:

```
P3(x+y, y+1, z+2);
P3(x, 2, z);
```

где x, y, z – глобальные переменные. Такие инструкции вызовут синтаксическую ошибку. В то же время синтаксически правильными будут вызовы

```
P3(x, y, z+1);
P3(x, y, 2);
```

При использовании параметров-значений во время вызова процедуры происходит присваивание значения формальному параметру, т. е. значение записывается в выделенные под хранение параметра ячейки памяти. При использовании параметров-переменных память под них не выделяется. Вместо этого на время работы процедуры они становятся синонимами тех переменных, которые указываются в качестве фактических параметров. Так, при вызове P3(x, y, 2) формальные параметры a и b будут работать с теми же ячейками памяти, что и переменные x, y. Соответственно если формальному параметру в процедуре присваивается новое значение, то и значение фактического параметра изменится. Воспроизведем пример, заменив тип параметров (добавим **var** перед их описанием):

```
var
   a, b: integer;    {глобальные переменные}
procedure P3(var a, b: integer);    {внутри процедуры
                      символами a и b обозначаются фор-
                      мальные параметры. Теперь это па-
                      раметры-переменные, и действия с
                      ними никак не повлияют на значе-
                      ния глобальных переменных a и b}
begin
   a:=a+1;
   b:=b+1;
   writeln(a+b);
end;

begin
   a:=1;
   b:=1;
```

```
 P3(a, b);          {вызов процедуры, где в качестве
                    фактических параметров использо-
                    ваны глобальные переменные a и b.
                    После вызова глобальные перемен-
                    ные увеличатся на 1}
   writeln(a, b);
end.
```

Если без слова **var** программа выводила числа 4, 1, 1, то после его добавления получится 4, 2, 2.

11.5. Программирование сверху вниз

Вряд ли стоило бы уделять много внимания процедурам, если бы за ними не скрывались важные и основополагающие идеи. В действительности процедуры оказывают решающее влияние на стиль и качество работы программиста. Процедура – это не только способ сокращения текста, но, что более важно, средство разложения программы на логически связанные, замкнутые компоненты, определяющие ее структуру.

Представьте себе программу, содержащую, например, 1000 строк кода (это еще очень маленькая программа). Обозреть такое количество строк и понять, что делает программа, было бы практически невозможно без процедур.

Большие программы строятся методом последовательных уточнений. На первом этапе внимание обращено на глобальные проблемы, и в первом эскизном проекте упускаются из виду многие детали. По мере продвижения процесса создания программы глобальные задачи разбиваются на некоторое число подзадач. Те, в свою очередь, на более мелкие подзадачи и т.д., пока решать каждую подзадачу не станет достаточно просто. Такая декомпозиция и одновременная детализация программы называется *нисходящим методом программирования*, или *программированием сверху вниз*.

Концепция процедур позволяет выделить отдельную подзадачу как отдельную подпрограмму. Тогда на каждом этапе можно придумать имена процедур для подзадач, вписать в раздел описаний их заголовки и, еще не добавляя к ним тело процедуры, уже использовать их вызовы для создания каркаса программы так, будто процедуры уже написаны.

Пример 6. Каркас программы сортировки массивов.

```
program ArraySort;
const
   n = 10;
type
   TArray = array [0..n-1] of real;
var
   a: TArray;
```

```
procedure InputArray(var x: TArray);  {процедура для
                ввода массива}
procedure PrintArray(x: TArray);  {процедура для выво-
                да массива}
procedure SortArray(var x: TArray);   {процедура сор-
                тировки}
begin
   InputArray(a);     {вызов процедуры ввода массива}
   SortArray(a);      {вызов процедуры сортировки}
   PrintArray(a);     {вызов процедуры вывода}
end.
```

Какие именно инструкции будут выполняться процедурами InputArray, PrintArray и SortArray, пока не определено. Это следующий этап написания программы. Задачу сортировки, в свою очередь, можно разбить на более простые подзадачи: многократное повторение поиска минимального элемента и постановку его на нужное место (см. параграф 10.5). Эти задачи также можно оформить в виде процедур, создав каркас процедуры SortArray.

Когда каркас создан, остается только написать тела процедур. Преимущество такого подхода в том, что, создавая тело каждой из процедур, можно не думать об остальных процедурах, сосредоточившись на одной подзадаче. Кроме того, когда каждая из процедур имеет понятный смысл, гораздо легче, взглянув на программу, понять, что она делает. Это позволит допускать меньше логических ошибок и организовать совместную работу нескольких программистов над большой программой.

Важной идей при таком подходе становится использование локальных переменных. В глобальную переменную можно было бы записать результат работы процедуры, однако это будет стилистической ошибкой. Весь обмен информацией процедура должна вести исключительно через параметры. Такой подход позволяет сделать решение каждой подзадачи максимально независимым от решения остальных подзадач, упрощает и упорядочивает структуру программы.

11.6. Передача массивов в качестве параметров

Как вы знаете, с отдельными элементами массивов можно работать так же, как с обычными (скалярными) переменными. Соответственно появляется возможность использовать их в качестве фактических параметров. Однако может потребоваться передать процедуре не отдельные элементы, а весь массив целиком. При этом недопустим такой, например, способ описания параметра:

```
procedure P(a: array [1..10] of integer);{так нельзя}
```

Тип параметра должен быть или скалярным, или заранее определенным в разделе **type**. Иными словами, правильным будет такое описание:

```
type
   TArray = array [1..10] of integer;
var
   x: TArray;
procedure P(a: TArray);
```

Такую процедуру P можно вызывать, указав в качестве фактического параметра, например, глобальную переменную x.

Пример 7. Программа, вычисляющая сумму элементов массива.

```
const
   n = 10;
type
   TArray = array [0..n-1] of integer;
var
   x: TArray;
   i, sum: integer;

procedure Summa(a: TArray; var s:integer);
var
   i: integer;
begin
   s:=0; {операции с параметром-переменной s отразят-
            ся на фактическом параметре – глобальной
            переменной sum}
   for i:=0 to n-1 do
      s:=s+a[i];
end;
begin
   for i:=0 to n-1 do
      readln(x[i]);
  Summa(x, sum); {к моменту вызова процедуры перемен-
           ной sum ничего еще не присвоили. Это не
           страшно, так как начальное значение пара-
           метра s для работы процедуры несущественно}
   writeln(sum);
end.
```

При вызове процедуры содержимое массива x будет скопировано в массив a. Глобальная переменная sum на время работы процедуры приобретет синоним s. Другими словами, присваивая что-то переменной s, мы тем самым изменяем и переменную sum. В итоге программа выведет сумму элементов введенного пользователем массива.

При передаче массива через параметр-значение происходит копирование его содержимого в новые ячейки памяти. При работе с массивами

большого размера или при необходимости вызывать процедуру очень много раз это может привести к большим затратам времени. Чтобы избежать этого, можно передавать массивы через параметры-переменные. При этом фактического переноса содержимого массивов из одной области памяти в другую не происходит, и вызов процедуры занимает меньше времени. Другими словами, при большом размере массива правильнее будет заголовок

```
procedure Summa(var a: TMassive; var s: integer);
```

11.7. Функции

Желаемым результатом работы подпрограммы может быть всего одно значение. Так, например, было в примере с расчетом суммы элементов массива. В этом случае вместо процедур разумно использовать другой вид подпрограмм, а именно функции.

Пример 8. Функция, вычисляющая сумму своих аргументов.

```
function Sum(a, b: real): real;
begin
   Sum := a + b;
end;
```

Мы видим, что структура функции почти повторяет структуру процедуры. Отличия:

1) вместо слова **procedure** пишется слово **function**;

2) после списка параметров через двоеточие указывается тип значения, которое получится в результате работы функции;

3) в теле функции должно присутствовать присваивание значения идентификатору функции (`Sum:=a+b`). Это и будет значением функции.

Вызов функции из программы происходит также, как и в случае со стандартными функциями, таким как `sqr`, `sqrt`, `random`, `sin`, `round` и т.д. Иными словами, их значения можно присваивать переменным или использовать их в выражениях. Например, в программе могли бы стоять вызовы:

```
x := sum(2, 2);
z := sum(x, y);
x := 2*sum(x, y)+1;
z := sqrt(sum(sqr(x), sqr(y)));
z := sum(sum(x, y), z);
```

и т.п.

С точки зрения синтаксиса допустим вызов функции без присваивания ее значения какой-нибудь переменной. Например, в программе могла бы быть строчка

```
sum(2, 2);
```

и все. Созданное функцией значение в этом случае никуда не запишется. Оно попросту пропадет. Впрочем, функция может сделать еще что-нибудь полезное, например изменить один из своих параметров-переменных или глобальную переменную. Так что такой процедуроподобный вызов может иметь смысл.

Правила передачи параметров те же самые, что и для процедур. Так же можно описывать локальные переменные.

Для примера приведем функцию, вычисляющую сумму элементов вещественнозначного массива.

Пример 9. Функция, вычисляющая сумму элементов массива.

```
const
   n = 10;
type
   TArray = array [0..n-1] of real;
function ArraySum(var a: TArray): real;
var
   s: real;
   i: integer;
begin
   s:=0;
   for i:=0 to n-1 do
      s:=s+a[i];
   ArraySum:=s;
end;
```

11.8. Опережающее описание

В теле каждой процедуры или функции может содержаться вызов других процедур или функций при условии, что они описаны раньше, чем процедура или функция, их вызывающая. Однако есть возможность вызывать и те процедуры, которые описаны после вызывающей. Для этого надо скопировать заголовок вызываемой подпрограммы и разместить его выше всех описаний. Например:

```
{Опережающее описание функции F1 - заголовок без тела функции}
function F(x: real): real;
procedure P;
var
   x: real;
begin
   ... {Какие-то действия}
   {Вызов функции F возможен благодаря опережающему описанию}
   x:=F(x);
```

```
   ... {Какие-то действия}
end;

{Описание функции F, теперь тело функции присутствует}
function F(x: real): real;
begin
   ... {Какие-то действия}
   F1 := … {Что-то}
end;
```

Это может понадобиться, когда есть две процедуры, вызывающие друг друга, например:

```
procedure A(n: integer); {опережающее описание первой процедуры}
procedure B(n: integer); {опережающее описание второй процедуры}
procedure A(n: integer);{полное описание процедуры A}
begin
   writeln(n);
   B(n-1);
end;
procedure B(n: integer);{полное описание процедуры B}
begin
   writeln(n);
   if n<10 then
      A(n+2);
end;
```

Процедура A вызывает процедуру B, та, в свою очередь, вызывает A, а та снова B и т.д. Данная цепочка вызовов закончится, поскольку B вызывается каждый раз с на единицу большим значением параметра n. Когда n перестанет быть меньше 10, процедура B завершится, не вызвав A, что позволит завершиться и прочим процедурам в цепочке.

Подобный прием в программировании называется *рекурсией*. Более подробно рекурсия будет рассмотрена нами позднее.

11.9. Процедурные типы

Значением переменной процедурного типа является процедура или функция. Например, в ситуации, когда нужно многократно повторять действия, совпадающие с точностью до замены какого-то выражения или небольшого набора операторов, это выражение или набор операторов можно передать в процедуру в качестве параметра процедурного типа.

Для объявления процедурного типа используется заголовок процедуры или функции без указания имени. Например:

```
type
   TProc1 = procedure (a, b, c: real; var d: real);
   TProc2 = procedure (var a, b: array of integer);
   TProc3 = procedure; {процедура без параметров}
   TFunc1 = function: real; {функция без параметров}
   TFunc2 = function (var x:array of integer): inte-
ger;
var
   Proc1: TProc1;
   Proc2: TProc2;
   Proc3: TProc3;
   Func1: TFunc1;
   Func2: TFunc2;
```

Если в программе описаны процедуры или функции с подходящим видом заголовка, то их можно присваивать переменным процедурного типа. Например, если есть функция с заголовком

```
function ArraySum(var a: array of integer):integer;
```

вычисляющая сумму элементов массива, то в программе допустимо присваивание:

```
Func2 := ArraySum;
```

После такого присваивания инструкция

```
s := Func2(x);
```

запишет в переменную s сумму элементов массива x (фактически будет вызвана функция ArraySum).

Пример 10. Интегрирование методом трапеций.

Рассмотрим пример задачи, где могут быть полезны процедурные типы. Пусть требуется создать функцию, производящую численное интегрирование любой функции одной переменной, т. е. найти значение

$$I = \int_a^b f(x)dx .$$

Воспользуемся так называемым *методом трапеций*. Для этого интервал [*a*, *b*] разобьем на *N* равных отрезков. Ширина каждого составит

$$h = \frac{b-a}{N}.$$

Концы отрезков будут иметь координаты

$$x_i = a + i \cdot h,\ i = 0..N .$$

Оценим интеграл как сумму площадей трапеций, построенных на основе этих отрезков (рис. 11.1).

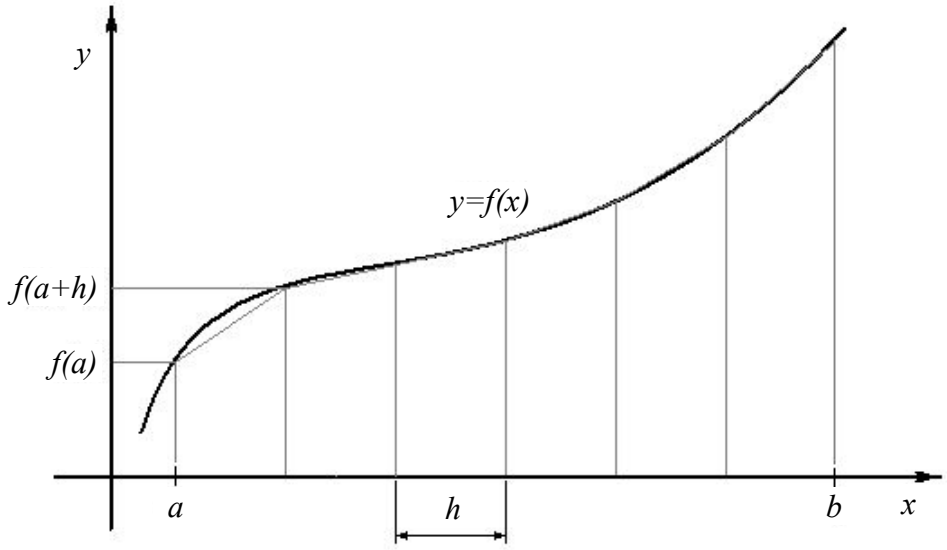


Рис. 11.1

Площадь *i*-й по счету трапеции составит

$$S_i = \frac{f(x_i) + f(x_{i+1})}{2} \cdot h .$$

Оценка интеграла, таким образом, дается формулой

$$\widetilde{I} = \sum_{i=1}^{N-1} S_i = \left(\frac{f(a) + f(b)}{2} + \sum_{i=1}^{N-2} f(x_i) \right) \cdot h .$$

Очевидно, чем меньше величина *h* (чем на большее число отрезков разбивается интервал интегрирования), тем точнее получится оценка интеграла.

Составим каркас универсальной функции, вычисляющей такой интеграл:

```
type
   TFunc = function (x: real): real; {процедурный тип
       – функция одной переменной}
function Example (x: real): real;
{Та функция, от которой впоследствии будем брать ин-
       теграл}
begin
   Example := sin(x);
end;
function Integral(a, b: real; f: TFunc): real;
begin
   … {реализация метода трапеций с использованием
       функции f, переданной через параметр процедур-
       ного типа}
end;
```

```
{Начало основной программы}
begin
   Int:=Integral(0, Pi, Example); {вызов функции In-
       tegral, где в качестве параметра передана
       функция Example, вычисляющая синус}
end.
```

Вместо Example могла бы стоять произвольная функция, описанная в программе.

Непосредственную реализацию метода трапеций выполните самостоятельно.

11.10. Правильное составление заголовков процедур и функций

Хорошо написанная подпрограмма должна быть максимально независима от остальной программы. Для этого в ней не должно быть обращений к глобальным переменным, а всю необходимую информацию она должна получать и отдавать через параметры. Составление заголовков (т. е., по сути, выбор того, что сделать параметрами), таким образом, требует понимания, какие данные необходимы для решения оформляемой в виде подпрограммы подзадачи (входные данные) и в каком виде лучше всего представить результат (выходные данные).

Простейший пример: требуется возвести число в квадрат. Какие данные для этого необходимы? Необходимо само число, которое будем возводить, и все. Что будет на выходе? Опять же одно число – квадрат того, что на входе. Данные на входе должны быть параметрами процедуры или функции (как правило, параметрами-значениями). Выходные данные необходимо записывать в параметры-переменные или в значение функции. Соответственно есть следующие варианты написания подпрограмм для решения этой задачи:

```
procedure NewSqr1(x: real; var x2: real);
{Входные данные передаются через параметр-значение x,
результат записывается в параметр-переменную x2}
begin
   x2:=x*x;
end;

function NewSqr2(x: real): real;
{Входные данные передаются через параметр-значение x,
результат записывается в значение функции}
begin
   NewSqr2:=x*x;
end;

procedure NewSqr3(var x: real);
```

```
{Одна и та же переменная служит для хранения входных
и выходных данных}
begin
   x:=x*x;
end;
```

Пусть есть две переменные:

```
a, b: real;
```

Записать в переменную *b* квадрат переменной *a* с помощью показанных выше процедур и функций можно следующими способами:

```
NewSqr1(a, b);
```

```
b:=NewSqr2(a);
```

```
b:=a;
NewSqr3(b);
```

Некоторые студенты помещают в заголовок описания всех переменных, которые им требуются внутри процедуры, явно путая назначение параметров и локальных переменных. **Так делать нельзя!**

Итак:

1) Параметрами должны быть только входные или выходные данные.

2) Все переменные, значение которых неизвестно или несущественно в момент начала работы процедуры и значения которых не важны после окончания работы, должны быть локальными переменными.

Пример: функция для расчета факториала. Что должно быть на входе? Целое число, факториал которого хотим подсчитать. Что на выходе? Значение факториала – тоже целое число. Таким образом, заголовок будет выглядеть так:

```
function Factorial(n: integer): integer;
```

При расчете потребуется переменная-счетчик цикла, а также переменная, в которой будет накапливаться произведение, но они должны быть локальными переменными.

Теперь пара слов, почему так важно правильно составлять заголовки, определяя формат данных на входе и выходе. Как уже говорилось, разработка любой сложной программы требует разбиения задачи на подзадачи, решение которых оформляется в виде процедур или функций. Далее программист сосредоточивается на решении отдельной подзадачи. Новичкам бывает трудно это сделать – начинаешь решать одну подзадачу, оказывается, что для этого требуется решить другую, потом третью и т.д. Так и пытаются решить задачу целиком, что, начиная с определенного уровня сложности, просто невозможно.

Выход из этой ситуации следующий. Если, решая одну задачу, вы видите, что требуется решить подзадачу, необходимо определить, какие

данные для этого необходимы и какие будут являться результатом расчетов. Затем вы можете составить заголовок процедуры или функции, решающей подзадачу. Имея заголовок, можно отложить написание тела подпрограммы на потом, а в программу вставить вызов процедуры или функции так, будто она уже написана. В результате вы решаете основную задачу, не отвлекаясь на возникающие по ходу подзадачи.

Последний вопрос. Имеется подзадача. Что писать, процедуру или функцию? Функция пишется тогда, когда результат имеет простой (не структурный) тип, т. е. не является массивом (а также файлом, записью, классом или множеством). Кроме того, возможен вариант, когда часть выходных данных передается через параметры, а часть записывается в значение функции.

Задача 1. Опишите вещественнозначный массив и вещественную переменную, напишите заголовок подпрограммы, подсчитывающий сумму элементов массива, и ее вызов с использованием описанных вами переменных.

Решение:

Задаем обязательные вопросы:

1) Что на входе?

Массив.

2) Что на выходе?

Сумма элементов – одно число. Входные данные передадим через параметр, выходные запишем в значение функции. Правильный ответ к задаче выглядит так:

```
const
   n = 10;
var
   a: array [0..n-1] of real;
   s: real;
   …
function Sum(var x: array of real): real;
…
s:=Sum(a);
```

Не будет ошибкой и оформление подпрограммы в виде процедуры с записью результата в параметр-переменную:

```
procedure Sum(var x: array of real; var s: real);
…
Sum(a, s);
```

Задача 2. Напишите заголовок подпрограммы для нахождения суммы элементов массива, индексы которых лежат в заданном диапазоне. Напишите также вызов этой подпрограммы, описав необходимые для этого переменные.

Решение:

Задаем обязательные вопросы:

1) Что на входе?

На входе обязательно массив, требуется также задать диапазон индексов. Для этого можно сказать, начиная с какого и по какой индекс будем суммировать (т. е. задать два целых числа).

2) Что на выходе?

Если удастся подсчитать сумму, то на выходе сумма элементов (т. е. число). Однако нелишним будет сделать «защиту от дурака». Что если в массиве 10 элементов, а при вызове требуется подсчитать сумму с 1-го по 20-й? Предупредить, что такое невозможно и элементов с такими индексами нет, можно с помощью переменной-флага (см. параграф 7.1) – логической переменной, которая равна true, если все в порядке, или false, если расчет невозможен. Таким образом, на выходе вещественное число и логическое значение.

Ответ: правильных вариантов масса. Например, оформляем решение в виде функции

```
const
   n = 10;
var
   a: array [0..n-1] of real;
   s: real;
   n1, n2: integer;
   f: boolean;
   …
function Sum(var x: array of real; n1, n2: integer;
var flag: boolean): real;
…
{Вызов функции}
s:=Sum(a, n1, n2, f);
```

Другой правильный вариант:

```
function Sum(var x: array of real; n1, n2: integer;
var s: real): boolean;
…
f:=Sum(a, n1, n2, s);
```

Еще правильный вариант:

```
procedure Sum(var x: array of real; n1, n2: integer;
var s: real; var f: boolean);
…
Sum(a, n1, n2, s, f);
```

Задача 3. Создать подпрограмму, увеличивающую все элементы массива в два раза.

Обязательные вопросы.

Что на входе?

Массив.

Что на выходе?

Такой же массив, увеличенный в 2 раза.

Ответ:

```
const
   n = 10;
var
   a: array [0..n-1] of real;
   …
procedure Double(var x: array of real);
…
Double(a);
```

Здесь один и тот же массив x используется как входная и выходная переменные.

Еще правильный вариант:

```
const
   n = 10;
var
   a, b: array [0..n-1] of real;
   …
procedure Double(x: array of real; var y: array of real);
…
Double(a, a);
```

Следующий вариант вызова не «портит» исходный массив:

```
Double(a, b);
```

Задания на составление заголовков

I. *Напишите заголовки подпрограмм для решения представленных ниже задач. Запишите также вызов этих подпрограмм в основной программе, описав необходимые для этого переменные (все по аналогии с приведенными выше примерами). Если есть возможность оформить подпрограмму и как процедуру, и как функцию, запишите ответ в обоих вариантах.*

1) Нахождение факториала.
2) Возведение вещественного числа в целую степень.
3) Расчет среднего арифметического элементов массива.
4) Перестановка элементов массива в обратном порядке.
5) Нахождение максимального элемента числового массива.
6) Нахождение индекса максимального элемента массива.

7) Нахождение индекса максимального элемента среди последних n элементов массива.

8) Нахождение индекса максимального элемента среди первых n элементов массива.

9) Нахождение расстояния между точками в многомерном пространстве.

10) Нахождение модуля вектора.

11) Нахождение скалярного произведения.

12) Нахождение суммы двух векторов.

13) Нахождение векторного произведения двух векторов. Векторное произведение может быть рассчитано только для 3-мерных векторов. Если вектор не 3-мерный, передавайте информацию о невозможности расчета через флаговую переменную.

14) Нахождение смешанного произведения.

15) Нахождение суммы двух комплексных чисел.

16) Нахождение угла между двумя векторами.

17) Вычисление гипотенузы по известным катетам.

18) В одном массиве содержатся фамилии, в другом – даты рождения соответствующих людей. Найти по фамилии год рождения.

19) При наличии тех же массивов, что и в предыдущей задаче, узнать, сколько человек родилось в заданном году.

20) При наличии тех же массивов распечатать фамилии всех людей, возраст которых лежит в заданном диапазоне.

II. *Придумайте задачу, для решения которой потребуются подпрограммы со следующими заголовками:*

```
procedure P1(a, b: integer);
procedure P2(var a, b: real);
procedure P3(a, b: real; var c: real);
function F4(x: real): real;
function F5(x: real): integer;
function F6(x, y: real): boolean;
function F7(x: array of real): real;
procedure P8(x, y: array of real; var z: array of real);
function F9(var s: array of string; var x: array of integer; z: integer; var flag: boolean): integer;
function F10(var s1, s2: array of string; var n: integer): boolean;
```

11.11. Модули

Разработка больших программ невозможна без их структурирования. Первым шагом в этом направлении было использование процедур и функций, в которые можно помещать решения отдельных подзадач. При этом часть программы и необходимые для ее работы переменные выделяются в

независимый (по крайней мере, к этому следует стремиться) от главной программы блок – подпрограмму. Следующим шагом в структурировании является объединение логически связанных подпрограмм, а также типов, переменных и констант в общую структуру – модуль.

Модуль – это отдельный файл, содержащий описание констант, типов, переменных, процедур и функций. Главная программа может использовать модуль, при этом описанные в модуле константы, типы и переменные становятся глобальными для программы, а процедуры и функции – доступными для вызова.

Чтобы подключить модуль к программе, используется ключевое слово **uses** (использует), после чего идет перечисление имен используемых программой модулей. Это подключение располагается в разделе описаний программы, традиционно в самом его начале. Например:

```
program P1;
uses {Ключевое слово, начинающее подключение модулей}
   Modul1, Modul2; {перечисление имен модулей}
var
   <и т.д.>
```

Модуль создается в отдельном pas-файле и имеет следующую структуру:

```
Unit <Имя модуля>;

interface

   <Интерфейсная часть>

implementation

   <Реализация или исполняемая часть>

begin
   <Инициирующая часть>
end.
```

Имя модуля произвольное, но должно совпадать с именем pas-файла, где модуль хранится.

В интерфейсной части располагается описание констант, типов и переменных, которые будут доступны главной программе после подключения модуля. Здесь также располагаются заголовки (только заголовки, без тел) процедур и функций, доступных главной программе. Полностью процедуры и функции прописываются в исполняемой части. Там же могут быть описаны константы, типы и переменные, которые будут глобальными для модуля, но не доступны в главной программе. Также в исполняемой части может располагаться описание процедур и функций, заголовки кото-

рых не помещены в интерфейсную часть. Они доступны для вызова другими процедурами и функциями модуля, но не доступны в главной программе.

И интерфейсная, и исполняемая часть могут содержать подключение других модулей.

В инициирующей части располагаются операторы, которые будут выполнены при подключении модуля к главной программе, т. е. еще до того, как начнут выполняться операторы главной программы. Например, в инициирующей части часто располагают присваивание начальных значений переменным. Инициирующая часть может отсутствовать.

Если вы работаете в среде Turbo Pascal, то после того как pas-файл модуля создан, его следует откомпилировать. Это делается нажатием клавиши F9. В результате будет создан файл с расширением tpu (Turbo Pascal Unit), который фактически и будет подключаться к программе.

Переключаясь с написания модуля на написание основной программы, обязательно сохранитесь. Если этого не сделать, к программе будет подключен последний сохраненный вариант модуля без окончательных внесенных вами изменений.

11.12. Хороший стиль при написании процедур и функций

Под стилем в программировании подразумеваются неформальные правила написания и оформления программ, нарушение которых допустимо, но, как показал многолетний опыт, крайне нежелательно. Без серьезной на то причины делать этого не следует.

Итак, при работе с процедурами и функциями необходимо придерживаться следующих правил:

1) Логически замкнутые куски программы следует оформлять в виде процедур и функций, даже если не предполагается выполнять заключенные в них инструкции более одного раза.

На будущее: с этого момента при решении каждой задачи хотя бы часть программы должна быть оформлена в виде процедуры или функции.

2) Если значение переменной используется при расчетах внутри подпрограммы, но не требуется за ее пределами, следует делать ее локальной.

3) Предыдущее правило можно расширить: необходимо избегать всякого обращения внутри процедур к глобальным переменным. Весь обмен данными с главной программой должен производиться через параметры и/или значения (для функций).

Следование этому правилу позволит создавать подпрограммы, работоспособность которых не зависит от остальной программы. Такие подпрограммы можно спокойно переносить из одной программы в другую. Они не начнут работать неправильно после внесения поправок в главную программу или другие подпрограммы.

4) **Все переменные-счетчики циклов, используемые в подпрограммах, должны быть локальными переменными.** В принципе это правило следует из двух предыдущих, но оно очень важно.

5) Если одним из параметров является массив большого размера, его следует сделать параметром-переменной. Это экономит память, а обращение к подпрограмме занимает меньше времени.

6) При написании заголовков в списке формальных параметров рекомендуется ставить пробелы после запятых (если есть перечисление идентификаторов через запятую), двоеточий (перед именем типа) и после символа точка с запятой (если есть параметры разных типов). Также не помешает ставить пробел между фактическими параметрами при вызове процедуры или функции. В результате программы будут лучше смотреться и легче читаться.

7) Имена процедурам и функциям рекомендуется давать такие, которые как-то связаны с выполняемыми ими действиями. Иными словами, Proc1 и Func1 – это, как правило, плохие имена. Нормальное имя имеет длину 5 – 15 символов – не стоит лениться или экономить место.

Контрольная работа 10

1. Напишите заголовки следующих подпрограмм:

а) процедуры, имеющей один целочисленный параметр-значение и один вещественный параметр-переменную;

б) процедуры, имеющей два вещественных параметра-переменных;

в) функции, имеющей два целочисленных аргумента и принимающей вещественные значения.

2. Что выведет программа?

```
var
   i: integer;
procedure Hello(n: integer);
var
   i: integer;
begin
   for i:=1 to n do
      write('Hello_');
   writeln;
end;
begin
   Hello(5);
   for i:=1 to 3 do
      Hello(i-1);
   Hello(3);
end.
```

3. Что выведет программа?

```
var
   a, b: integer; {Глобальные переменные}
procedure P1(a, b: integer);
begin
   a:=a+1;
   b:=2*b+1;
   writeln(a-b);
end;
procedure P2(var a, b: integer);
begin
   a:=a+1;
   b:=2*b+1;
   writeln(a-b);
end;
procedure P3(var a: integer; b: integer);
begin
   a:=a+1;
   b:=2*b+1;
   writeln(a-b);
end;
procedure P4(a: integer; var b: integer);
begin
   a:=a+1;
   b:=2*b+1;
   writeln(a-b);
end;
begin
   a:=1;
   b:=1;
   P1(a, b);
   P1(b, a);
   P1(a, a);
   P1(2*b, b-a);
   P1(a+1, a-1);
   P1(a+1, 5);
   P1(0, 0);
   P1(100, 100);
   writeln(a, b);
   P2(a, b);
   P2(b, a);
   P2(a, a);
   P2(b, b);
   writeln(a, b);
   P3(a, b);
   P3(b, a);
```

```
   P3(a, a);
   P3(b, b);
   P3(a, a+1);
   P3(b, a+1);
   writeln(a, b);
   P4(a, b);
   P4(b, a);
   P4(a, a);
   P4(b, b);
   P4(a+1, b);
   writeln(a, b);
end.
```

4. Что выведет программа?

```
const
   n = 5;
type
   TIntArray = array [0..n-1] of integer;
var
   x: TIntArray;
   i: integer;
procedure P(var x: TIntAray; var m:integer);
var
   i: integer;
begin
   m := 0;
   for i:=1 to n-1 do
      if x[i] > x[m] then
         m := i;
end;
begin
   for i:=0 to n-1 do
      x[i]:=(i+1) mod 3;
   for i:=0 to n-1 do
   begin
      P(x, m);
      x[m]:=0;
   end;
   for i:=1 to n do
      writeln(x[i]);
end.
```

5. Что выведет программа?

```
const
   n = 5;
```

```
type
   TIntArray = array [0..n-1] of integer;
var
   x, y: TIntArray;
   i: integer;
procedure Fill(var x: TIntArray);
var
   i: integer;
begin
   for i:=0 to n-1 do
      x[i] := i + 1;
end;
procedure P(var x, y: TIntArray);
var
   i: integer;
begin
   for i:=0 to n-1 do
   begin
      y[i] := y[i] + 1;
      x[i] := x[i] + y[n-i+1];
   end;
end;
begin
   Fill(x);
   Fill(y);
   P(x, y);
   P(y, x);
   for i:=0 to n-1 do
      writeln(x[i]);
end.
```

6. Что выведет программа?

```
var
   x, y, z: integer;
procedure Step(var x, y: integer);
begin
   if x>y then
      x := x - y
   else
      y := y - x;
end;
function Nod(x, y: integer): integer;
begin
   while x <> y do
      Step(x, y);
```

```
    Nod:=x;
end;
begin
    x:=9;
    y:=27;
    z:=3;
    writeln(Nod(Nod(x, y), z));
end.
```

Задание 11. Процедуры и функции

Простейшие процедуры

1. Создайте процедуру, печатающую на экране слово *Hello*, и программу, которая с ее помощью напечатает слово *Hello* 10 раз.

Локальные переменные

2. Создайте процедуру, которая выводит в одну строку слово *Hello* 5 раз. Осуществите это с помощью цикла. Переменную-счетчик цикла сделайте локальной. С помощью данной процедуры выведите 5 таких строк.

Параметры процедур

3. Создайте процедуру, печатающую слово *Hello* заданное число раз. Количество раз передавайте в процедуру как параметр-значение.

4. Создайте процедуру, увеличивающую значение переменной на единицу.

5. Создайте процедуру, меняющую значения двух переменных местами.

6. Создайте процедуру, располагающую два числа по возрастанию. На входе она должна получать две переменные, и если первая больше второй, то значения должны поменяться местами. Операцию перестановки выполняйте, обращаясь к процедуре 1.

7. Создайте процедуру, упорядочивающую три числа по возрастанию. Сравнений в ней не проводите. Вместо этого обращайтесь к процедуре из предыдущего задания.

8. Создайте процедуру, которая пару последовательных чисел Фибоначчи преобразует в следующую пару. Иными словами, если на входе даны элементы с номерами (n–1) и n, то процедура должна в те же переменные записать элементы с номерами n и (n+1). С помощью данной процедуры найдите 10-е число в последовательности Фибоначчи.

9. Создайте процедуру для возведения числа в целую степень. Число и степень должны быть параметрами-значениями, а результат должен записываться в параметр-переменную. Другими словами, заголовок процедуры должен выглядеть примерно так:

```
procedure Power(x: real; n: integer; var Result: real);
```

10. Поворот на угол φ против часовой стрелки относительно начала координат приводит к следующему преобразованию координат:

$$\begin{cases} x' = x\cos\varphi - y\sin\varphi, \\ y' = x\sin\varphi + y\cos\varphi. \end{cases}$$

Создайте процедуру, осуществляющую такое преобразование.

11. Создайте процедуру, заполняющую массив случайными числами. Массив для заполнения передавайте в процедуру как параметр-переменную. В той же программе создайте процедуру, печатающую на экране элементы массива. Главная программа должна состоять из вызова этих процедур.

12. Усовершенствуйте предыдущую программу, добавив к процедурам еще один параметр – количество элементов, которые нужно заполнять или печатать. Пусть главная программа запрашивает это количество у пользователя.

Функции

13. Создайте функцию, перемножающую два числа. Убедитесь с ее помощью, что дважды два – четыре.

14. Создайте функцию, возводящую число в целую степень. Число и степень должны быть, естественно, параметрами.

15. Создайте функцию, возвращающую индекс максимального элемента массива. С ее помощью осуществите сортировку массива.

16. Реализуйте не сделанные вами ранее задания на обработку массивов, оформляя часть программы как процедуру или функцию.

Процедурные типы

17. Создайте функцию, вычисляющую определенный интеграл методом трапеций. Функцию, от которой надо брать интеграл, передавайте как параметр процедурного типа.

12. Двумерные массивы

12.1. Двумерные массивы: теория

Если элементы одномерного массива можно мыслить себе как координаты некоторого вектора, то двумерный массив будет соответствовать матрице. Опишем для примера двумерный массив размером 10x20:

```
const
   n = 10;
   m = 20;
type
   TMatrix = array [0..n-1, 0..m-1] of real;
var
   A: TMatrix;
```

При обращении к элементам такого массива необходимо указывать сразу два индекса. Например, A[0,0] – левый верхний элемент матрицы и т.п. В остальном двумерные массивы ничем не отличаются от одномерных.

При желании можно вводить трех-, четырехмерные и т. д. массивы. Для этого при описании соответствующего типа следует указать не два, а еще больше диапазонов индексов.

Еще один способ описать двумерный массив, это описать массив массивов:

```
type
   TVector = array [0..n-1] of real;
   TMatrix2 = array [0..m-1] of TVector;
```

Каждый элемент массива TMatrix2 можно мыслить как вектор-строку матрицы. Обращение к левому верхнему элементу в этом случае будет выглядеть так: A[0][0]. Здесь A[0] – первая строка, которая сама является массивом, но одномерным.

Можно пользоваться любым из приведенных способов описания двумерного массива. Второй позволяет выполнять какие-то операции со строками целиком, а не поэлементно. Это же свойство можно посчитать недостатком из-за неравноценности строк и столбцов.

Задание 12. Двумерные массивы

1. Опишите тип – двумерный массив, количество элементов по горизонтали и вертикали пусть задается константами. Создайте процедуры: заполняющую такой массив случайными числами и печатающую массив на экране.

2. Создайте процедуры:

а) обнуляющую двумерный массив;

б) заносящую в квадратный двумерный массив единичную матрицу (с единицами на главной диагонали и нулями во всех прочих местах).

3. Создайте процедуру, присваивающую элементам двумерного массива их порядковые номера. Элементы нумеруются следующим образом:

а) построчно, т. е.

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & n \\ n+1 & n+2 & n+3 & \cdots & 2n \\ 2n+1 & 2n+2 & 2n+3 & & 3n \\ \vdots & \vdots & \vdots & & \vdots \\ (n-1)n+1 & \cdots & \cdots & \cdots & n^2 \end{pmatrix};$$

б) по спирали. Например, для матрицы 5×5 должно получиться

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 16 & 17 & 18 & 19 & 6 \\ 15 & 24 & 25 & 20 & 7 \\ 14 & 23 & 22 & 21 & 8 \\ 13 & 12 & 11 & 10 & 9 \end{pmatrix};$$

в) по диагонали. Например:

$$\begin{pmatrix} 1 & 3 & 6 & 10 & 15 \\ 2 & 5 & 9 & 14 & 19 \\ 4 & 8 & 13 & 18 & 22 \\ 7 & 12 & 17 & 21 & 24 \\ 11 & 16 & 20 & 23 & 25 \end{pmatrix}.$$

4. Создайте процедуру, которая вычтет строку с заданным номером, помноженную на коэффициент, из всех последующих строк матрицы. Матрица, номер вычитаемой строки и коэффициент должны быть параметрами.

5. Создайте процедуру, осуществляющую транспонирование матрицы.

13. Графика в Паскале

13.1. Введение

Набор графических средств в Паскале чрезвычайно ограничен и не годится для освоения современной компьютерной графики. Однако те элементарные операции, которые все же присутствуют, являются основой, на которой построены многие более сложные алгоритмы, хотя бы поэтому эти операции заслуживают изучения.

Для нас графические задачи будут важны еще и в связи с тем, что требуемый результат работы программы здесь легко понятен и нагляден. Поэтому они хорошо подходят для развития азов алгоритмического мышления.

13.2. Инициализация графического режима

Для того чтобы использовать в программе графические процедуры, необходимо подключить стандартный модуль с графическими процедурами и функциями. В среде PascalABC он называется GraphABC:

```
uses
   GraphABC;
```

Если вы используете PascalABC, то для перехода в графический режим этого достаточно.

Особенности диалекта Borland Pascal

В Borland Pascal соответствующий модуль называется просто Graph. Кроме того, для работы с графикой придется произвести ряд шаманских действий, а именно описать две целочисленные переменные (традиционно для них используют идентификаторы gd и gm):

```
gd, gm: integer;
```

В самой программе, перед тем как использовать графические процедуры, необходимо разместить такие строки:

```
gd:=DETECT;
InitGraph(gd, gm, '<путь к каталогу, где лежит файл egavga.bgi>');
```

Файл egavga.bgi обычно лежит в каталоге BGI, который, в свою очередь, находится в каталоге Паскаля. Например, путь может выглядеть так: C:\BP\BGI. Нет смысла подробно разбирать, что происходит при выполнении перечисленных операторов и что это за загадочный файл egavga.bgi. Паскаль, как известно, рассчитан на операционную систему MS DOS. Современные операционные системы с самого начала работают в графическом режиме, поэтому каких-то особых предварительных действий для работы с графикой не требуется (либо они будут совершенно другими). Так что просто вставляйте в программы указанные строки и не забивайте голову лишней информацией.

Необходимо иметь в виду, что после перехода в графический режим (после выполнения процедуры InitGraph) перестают работать процедуры текстового ввода/вывода (Read, Write и т.д.). Следовательно, если требуется что-то ввести с клавиатуры, делайте это до вызова InitGraph. Выйти из графического режима можно с помощью процедуры CloseGraph.

Таким образом, программа, работающая с графикой, будет выглядеть примерно следующим образом:

```
program GraphicsPrg;
uses
   Graph;
var
   gd, gm: integer;
   <Описание прочих переменных>
begin
   <Работа в текстовом режиме. Можно пользоваться Read'ом и Write'ом>
   gd:=DETECT;
   InitGraph(gd, gm, 'c:\BP\bgi');
   <Работа в графическом режиме. Можно рисовать>
   CloseGraph;     {выход из графического режима}
   <Снова работаем в текстовом режиме>
end.
```

13.3. Экранные координаты. Точка заданного цвета

При запуске программы с подключенным модулем GraphABC создается так называемое графическое окно, обычное для ОС Windows, в пределах которого можно рисовать. Площадь окна можно рассматривать как двумерный массив светящихся точек (пикселей). Координаты точек принимают целочисленные значения и отсчитываются от левого верхнего угла окна (рис. 13.1). Такой способ называется экранной системой координат. Ось ординат направлена вниз. Соответственно сам левый верхний угол имеет координаты (0,0). Координаты соседних с ним точек, очевидно, будут (0, 1), (1, 0) и (1, 1).

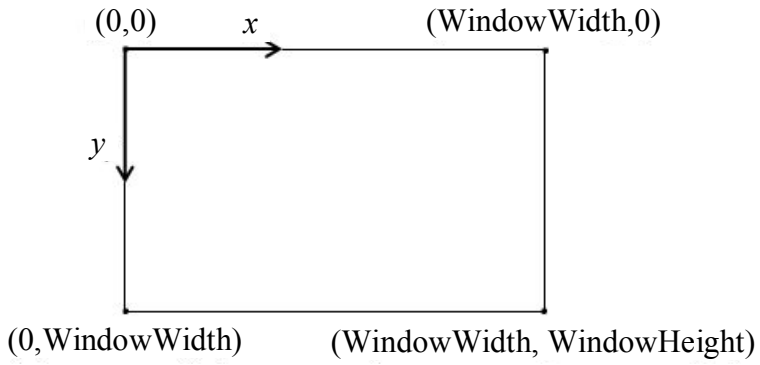


Рис. 13.1

Размеры окна по умолчанию составляют 640 на 480 точек. Однако пользователь может менять их произвольным образом обычным для ОС Windows способом (хватать мышью за угол, распахивать на весь экран и т.д.) Узнать текущий размер окна можно с помощью функций WindowWidth и WindowHeight. Например, инструкция

```
W := WindowWidth;
```

запишет текущую ширину в переменную W.

Изначально все пиксели окна имеют белый цвет. Однако каждому можно назначить произвольный цвет, заданный в так называемой системе RGB (расшифровывается как Red Green Blue). В этой системе цвет задается тремя целыми числами в диапазоне от 0 до 255, определющими интенсивность красной, зеленой и синей составляющей (как известно, любой цвет можно получить их смешением). Цветовые значения имеют специальный тип Color, их можно получить с помощью функции RGB, имеющей заголовок

```
function RGB(r, g, b: byte): Color;
```

Также цвет можно задать с помощью одной из определенных в модуле GraphABC констант. Например, clRed – красный цвет, clYellow – желтый и т.д. Полный список смотрите в справочной системе среды PascalABC (http:// pascalabc.net/ downloads/ pabcnethelp/ PABCUnits/ GraphABC/ gr_Colors.htm).

Указать любой точке, какого она должна быть цвета, можно с помощью процедуры PutPixel. Ее заголовок

```
procedure PutPixel(x, y: integer; c: Color);
```

Здесь x, y – координаты точки, c – цвет точки.

В принципе этой процедурой можно было бы и ограничиться. Раз для любой точки окна можно указать любой цвет, значит, можно нарисовать все, что угодно. Однако все же полезно иметь в своем распоряжении несколько дополнительных команд для рисования простейших объектов.

Особенности среды Borland Pascal

Ширина и высота экрана в пикселях здесь фиксирована (640 x 480).

По умолчанию все пиксели имеют черный цвет (не светятся), но каждому можно назначить один из 16 цветов. Каждому цвету соответствует целочисленная константа, описанная в модуле Graph. Список констант и их значения приведены в табл. 2.

Таблица 2

Цвет	Константа, описанная в модуле Graph	Значение константы
Черный	Black	0
Синий	Blue	1
Зеленый	Green	2
Голубой	Cyan	3
Красный	Red	4
Фиолетовый	Magenta	5
Коричневый	Brown	6
Светло-серый	LightGray	7
Темно-серый	DarkGray	8
Ярко-синий	LightBlue	9
Ярко-зеленый	LightGreen	10
Ярко-голубой	LightCyan	11
Ярко-красный	LightRed	12
Ярко-фиолетловый	LightMagenta	13
Желтый	Yellow	14
Белый	White	15

Произвольные цвета, не входящие в эту палитру, задать невозможно.

Назначение цвета производится с помощью той же процедуры PutPixel. Ее заголовок

```
procedure PutPixel(X, Y: integer; Color: Word);
```

Здесь `X, Y` – координаты точки, `Color` – цвет точки.

13.4. Простейшие графические объекты

Перечислим несколько процедур для рисования простейших графических объектов – так называемых *графических примитивов*:

– рисование линии:

```
procedure Line(X1, Y1, X2, Y2: integer);
```

Здесь `(X1, Y1)` – координаты начала, `(X2, Y2)` – координаты конца. Линия вычерчивается так называемым текущим цветом;

– установка текущего цвета:

```
procedure SetPenColor(C: Color);
```

Здесь `C` – константа (или число), задающая цвет. О задании цвета читайте в предыдущем параграфе;

– рисование окружности:

```
procedure Circle(X, Y: integer; R: Word);
```

Здесь `(X, Y)` – координаты центра, `R` – радиус окружности. Окружность, как и линия, рисуется текущим цветом;

– рисование дуги:

```
procedure Arc(X, Y, R, BegA, EndA: integer);
```

Здесь `(X, Y)` – координаты центра дуги, `BegA` и `EndA` – начальный и конечный углы дуги, `R` – радиус дуги. Углы отсчитываются против часовой стрелки от положительного направления оси абсцисс и указываются в градусах;

– вывод текста:

```
procedure OutText(X, Y: integer; Txt: string);
```

Данная процедура размещает текст `Txt` в прямоугольнике, левый верхний угол которого задается координатами `(X, Y)`;

– очистка экрана:

```
procedure ClearWindow;
```

За информацией о прочих графических примитивах отсылаем вас к справочной системе среды программирования (F1).

Особенности языка Borland Pascal

Некоторые процедуры в Borland Pascal имеют чуть другое название или другой порядок следования параметров:

– установка текущего цвета:

```
procedure SetColor(C: Word);
```

– рисование дуги (другой порядок следования параметров):

```
procedure Arc(X, Y: integer; BegA, EndA, R: Word);
```

– вывод текста:

```
procedure OutTextXY(X, Y: integer; Txt: string);
```

– очистка экрана:

```
procedure ClearDevice;
```

13.5. Экранный указатель

Экранный указатель – это аналог курсора в текстовом режиме, он указывает положение точки, которая считается «текущей». В отличие от мигающего курсора экранный указатель невидим. После инициализации графического режима экранный указатель располагается в точке с координатами (0, 0). Переместить его в произвольную точку на экране можно с помощью процедуры MoveTo. Ее заголовок

```
procedure MoveTo(X, Y: integer);
```

Координаты экранного указателя используются многими графическими процедурами. Например, процедура LineTo, имеющая заголовок

```
procedure LineTo(X, Y:integer);
```

рисует линию от экранного указателя до точки с координатами (X, Y). При этом экранный указатель смещается в точку (X, Y), что делает эту процедуру удобной для рисования ломаных линий, когда следующий отрезок начинается из конца предыдущего.

13.6. Мировые координаты

В задаче, решаемой программистом, может присутствовать своя собственная система координат. Например, если нужно построить график функции $y = \sin(x)$ на отрезке $[0, 2\pi]$, то координаты на плоскости, где мыслится этот график, очевидно, не совпадут с экранными. Требуется преобразовать координаты точек на графике в координаты изображаемых точек на экране. Такая система координат, присущая решаемой задаче, называется *мировой*.

Преобразование из мировых координат (x_m, y_m) в экранные (x_s, y_s) заключается в перемасштабировании (например, отрезок длиной 2π должен быть преобразован в отрезок длиной 600 пикселей) и сдвиге (например, точек с отрицательными координатами на экране нет и требуется сдвинуть перемасштабированный отрезок в положительную область). Это достигается путем линейного преобразования:

$$x_s = ax_m + b,$$

$$y_s = cy_m + d.$$

Здесь коэффициенты a и c отвечают за перемасштабирование (растягивание по горизонтали и вертикали), а b и d – за сдвиг вдоль этих направлений. Чтобы определить конкретные значения этих коэффициентов, надо решить, какую область (какой прямоугольник) в мировых координатах в какой области экрана мы собираемся отобразить.

Пусть мировые координаты меняются в диапазоне

$$x_{\min} \le x_m \le x_{\max},$$

$$y_{\min} \le y_m \le y_{\max}.$$

Область на экране, где требуется поместить изображение, зададим, указав координаты верхней левой точки (Left, Top), а также ширину и высоту этой области (Width и Height). Нетрудно убедиться, что в таком случае преобразование мировых координат в экранные будет выглядеть как

$$x_S = \mathrm{Left} + \frac{\mathrm{Width}}{x_{\max} - x_{\min}}(x_m - x_{\min}),$$

$$y_S = \mathrm{Top} + \mathrm{Height} - \frac{\mathrm{Height}}{y_{\max} - y_{\min}}(y_m - y_{\min}).$$

Данные преобразования дают в общем случае вещественный результат. Поскольку графические процедуры требуют целых значений аргумента, полученные по этим формулам координаты следует округлить.

Обратные преобразования, позволяющие по координатам точки на экране найти соответствующие мировые координаты, получите самостоятельно.

13.7. Сдвиг и поворот

Сдвиг на вектор с координатами (dx, dy) заключается в добавлении этих координат к координатам каждой точки сдвигаемого объекта:

$$x'_S = x_S + dx,$$

$$y'_S = y_S + dy.$$

Зачастую удобно задавать вектор сдвига его длиной и углом поворота (r, φ), что, как не трудно заметить, означает сдвиг на вектор с координатами

$$dx = r\cos\varphi,$$

$$dy = r\sin\varphi.$$

Поскольку речь идет о сдвиге точки на экране, то для ее прорисовки полученные значения надо округлить до целых. Однако если преобразование сдвига выполняется неоднократно или за сдвигом следуют другие преобразования (например, поворот), то округления, если они делаются после каждого преобразования, будут приводить к накоплению ошибок. Поэтому если координаты преобразуются несколько раз, лучше хранить их в вещественных переменных, а округление делать только непосредственно перед рисованием.

Преобразование сдвига, записанное для координат одной точки, можно применять для сдвига любой фигуры. Так, если надо сдвинуть линию, достаточно применить это преобразование к координатам обоих ее концов. Для сдвига окружности преобразование применяется к координатам ее центра и т.д.

Координаты точек при повороте на угол φ относительно начала координат преобразуются по закону (рис. 13.2):

$$x'_s = x_s \cos\varphi - y_s \sin\varphi,$$
$$y'_s = x_s \sin\varphi + y_s \cos\varphi.$$

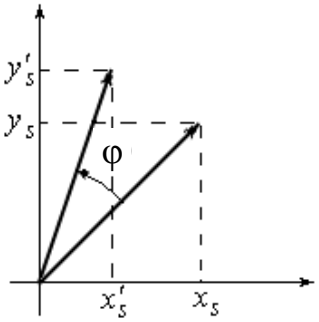


Рис. 13.2

Или в матричном виде

$$\begin{pmatrix} x'_s \\ y'_s \end{pmatrix} = \begin{pmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{pmatrix} \begin{pmatrix} x_s \\ y_s \end{pmatrix}.$$

Чтобы осуществить поворот относительно произвольной точки (x_0, y_0), необходимо сделать сдвиг, который перенесет эту точку в начало координат ($dx = -x_0$, $dy = -y_0$). После этого надо сделать поворот на требуемый угол и снова сделать сдвиг, который вернет бывшую точку (x_0, y_0) на место (рис. 13.3).

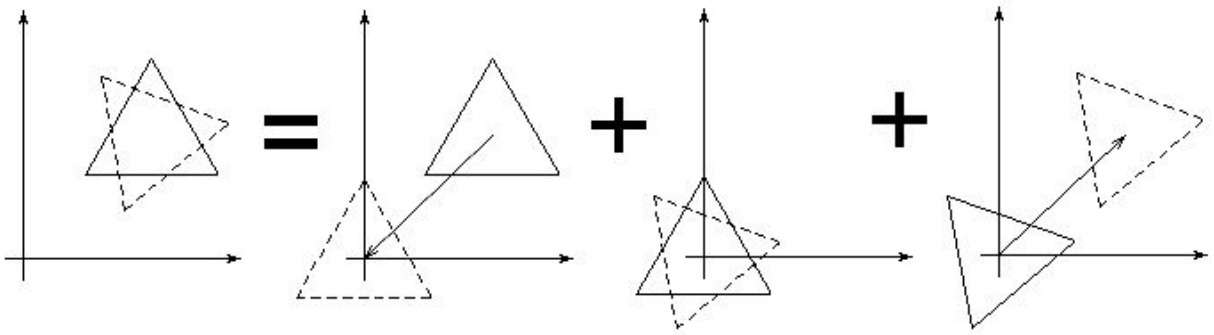


Рис. 13.3

Нетрудно записать такое преобразование следующим образом:

$$x'_s = (x_s - x_0)\cos\varphi - (y_s - y_0)\sin\varphi + x_0,$$
$$y'_s = (x_s - x_0)\sin\varphi + (y_s - y_0)\cos\varphi + y_0.$$

Задание 13. Графика в Паскале

Примечание: При выполнении заданий в среде Borland Pascal следует слово «графическое окно» заменить на «экран». Иными словами, вместо «нарисуйте в центре графического окна» следует читать «нарисуйте в центре экрана» и т.п.

1. Войдя в графический режим, определите ширину и высоту графического окна в пикселях. Выведите ее на экран, не выходя из графического режима.

2. Нарисуйте окружности радиусом 20 пикселей в центре графического окна и по углам окна.

3. С помощью процедур рисования линий изобразите квадрат со стороной 200 пикселей так, чтобы он отображался по центру окна.

4. Создайте процедуру, рисующую квадрат заданной величины по центру окна.

5. Отобразите на экране 1000 точек со случайными координатами и со случайными координатами, лежащими внутри квадрата из предыдущей задачи.

Усложненный вариант: со случайными координатами, лежащими внутри заданной окружности.

6. Воспроизведите орнаменты, оформив рисование их отдельных элементов в виде процедур (рис. 13.4).

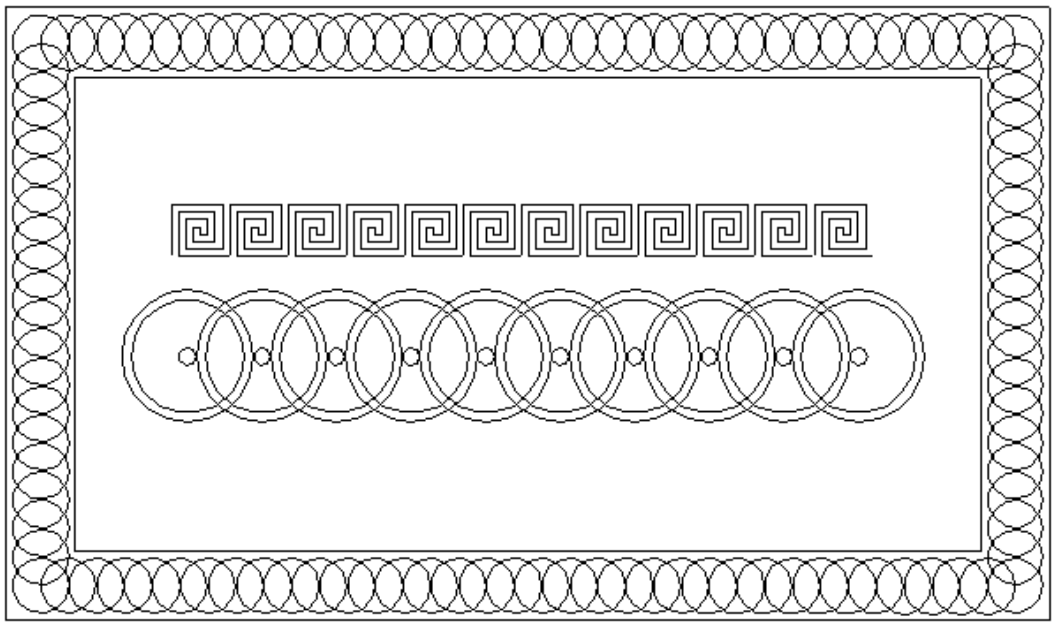


Рис. 13.4

7. Нарисуйте домик, над которым светит солнышко.

Усложненный вариант: (а) солнышко имеет лучики, (б) солнышко движется по небу; картинка рисуется процедурой, параметрами которой является координаты солнышка.

8. Воспроизведите рисунки, полученные с помощью поворотов треугольника и окружности (рис. 13.5).

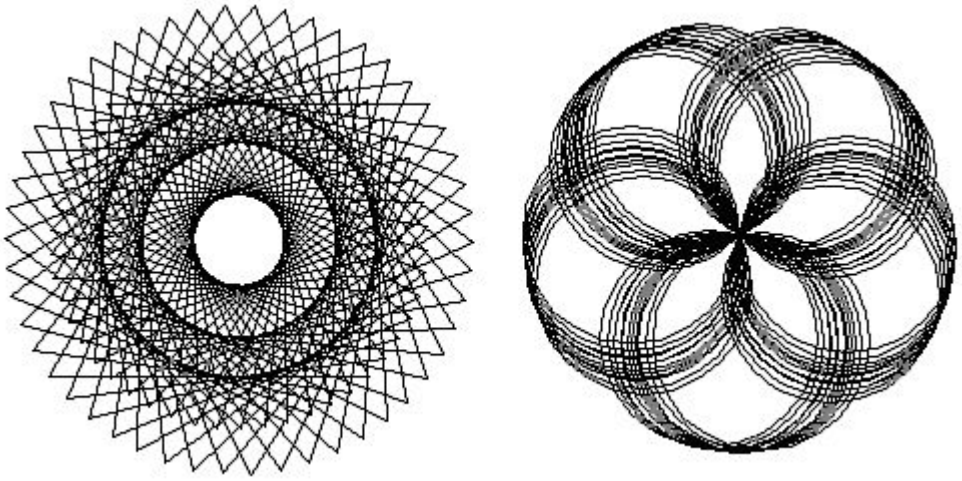


Рис. 13.5

9. Нарисуйте график функции $y = a\sin\omega t$. Параметрами процедуры должны быть интервал значений *t*, амплитуда *a*, частота ω и количество точек, выводимых на график.

14. Символы, строки, текстовые файлы

14.1. Символьный тип данных

Переменные типа *Char* (от английского *character*) могут хранить одиночный символ. В памяти такая переменная занимает 1 байт, соответственно она может принимать 256 различных значений (на самом деле в памяти хранятся коды символов – числа от 0 до 255). Какие именно символы соответствуют кодам, зависит от кодовой таблицы, установленной в операционной системе. Однако традиционно первые 128 из них – это так называемые ASCII-символы (см. http://ru.wikipedia.org/wiki/ASCII). Первые 32 символа называются управляющими, остальные изображаемыми. Управляющие символы воспринимаются устройствами вывода как команды. Например (табл. 3):

Таблица 3

Код	Действие	Английское название
7	Подача звукового сигнала	Beep
10	Признак конца строки текстового файла	End Of Line (EOLn)
13	Перевод строки	Line Feed (LF)
26	Признак конца файла	End Of File (EOF)

Опишем переменную символьного типа:

```
var
   c: char;
```

Есть два способа задать символьную константу. Первый – это написать символ в кавычках, например:

```
c := 'a';
c := '*';
```

и т.д.

Второй – это записать символ «решетка», за которым следует код задаваемого символа. Например:

```
c := #97;     {символ 'a'}
c := #7;      {подача звукового сигнала}
```

Заметим, что в данном случае речь идет только о символьных константах, и если код записан в целочисленную переменную, то написать #<имя переменной> нельзя.

Символы можно сравнивать. Больше тот символ, код которого больше. Соответственно 'a' < 'z', 'A' < 'Z' и '0' < '9'.

С символьным типом связаны следующие функции:

1) chr(i) – дает символ с кодом i. Вместо i может стоять любое выражение с целочисленным значением из диапазона от 0 до 255;

2) ord(c) – по символу определяет его код (функция обратная к chr);

3) succ(c) – символ, следующий в кодовой таблице за символом c;

4) pred(c) – символ, предшествующий в кодовой таблице символу c;

5) upcase(c) – преобразует строчные латинские буквы в прописные. Остальные символы оставляет неизменными, т. е. upcase('f') = 'F', а upcase('*') = '*'.

14.2. Строковый тип

Значениями строкового типа (string) в Паскале являются последовательности символов длиной от 0 (пустая строка) до 255 символов. Можно описать строку, предельная длина которой меньше 255, задав ее длину в квадратных скобках. Примеры описания строк:

```
var
   s1, s2: string; {обычные строки длиной до 255 сим-
                    волов}
   name: string[20]; {строка длиной не более 20 сим-
                    волов}
   group: string[3]; {строка длиной не более 3 симво-
                    лов}
   txt: array [0..99] of string[80]; {массив из 100
                    строк длиной не более 80 симво-
                    лов}
```

Строку можно рассматривать как массив символов, т. е. обращаться из программы к отдельным символам строки:

s1[2] – второй символ строки s1;

s2[i] – i-й символ строки s2;

txt[3][10] – десятый символ третьей строки из описанного выше массива txt.

Нумерация символов в строке начинается с 1. В памяти строка занимает на 1 байт больше, чем это необходимо для хранения символов. Самый первый (нулевой) байт хранит длину строки. Поэтому присваивание вида s1[10]:= 'x' приведет к желаемому результату, только если в строке десять или больше символов. Та же операция по отношению к более короткой строке не приведет к ее изменению.

Строки можно считывать и печатать обычными процедурами ввода/вывода: read, readln, write, writeln.

Для строк определена операция сложения. При этом складываемые строки объединяются в одну. Например:

```
s1 := 'abra';
s1[1] := Upcase(s1[1]);
```

```
s2 := 'kadabra';
s1 := s1+' '+s2+'!';
writeln(s1);
```

Здесь первый символ строки s1 (строчная «а») был заменен на прописную «А». Затем к этой строке были добавлены пробел и строка s2. В итоге программа напечатает строку 'Abra kadabra!'.

Переменную или выражение типа Char также можно прибавить к строке.

Отдельные символы строки (s1[1], s1[2] и т.д.) можно рассматривать как переменные типа Char. В примере мы воспользовались этим, заменив строчную букву заглавной.

Строка может быть пустой, т. е. вовсе не содержать символов и иметь нулевую длину. Такая строка задается как две одинарные кавычки, между которым ничего нет:

```
S := '';
```

Обратите внимание, что «ничего» в данном случае значит совсем ничего. Часто, пытаясь задать пустую строку, ставят между кавычками пробел, что неправильно.

Некоторые процедуры и функции для работы со строками:

1) Length(s) – получение длины строки;

2) Pos(Substr, S) – функция, в качестве результата выдающая номер символа, начиная с которого в строке s начинается подстрока Substr. Если S не содержит подстроки Substr, то результат будет равен 0.

Например: Pos(' ', s) – найдет номер позиции, на которой в строке s находится пробел. Pos('Ivan', s) найдет имя 'Ivan';

3) Copy(s, i, n) – функция, выделяющая из строки s подстроку, начиная с символа за номером i, и включающую n символов. Например:

```
s1 := 'abracadabra';
s1 := Copy(s1, 5, 6);
```

В результате строка s1 будет равна 'cadabr';

4) Delete(s, i ,n) – процедура, удаляющая из строки s n символов, начиная с символа с номером i;

5) Val(s, n, code) – процедура, переводящая строку s в число n. Если преобразование произошло успешно, то code будет равна 0, если нет, то в эту переменную будет записан код ошибки;

6) Str(n, s) – процедура, переводящая число n в строку s;

7) для преобразования чисел в строки и обратно также удобно пользоваться функциями:

– StrToInt(s) – функция, возвращающая целочисленное значение, записанное в строке s,

– StrToFloat(s) – функция, возвращающая целочисленное значение, записанное в строке s,

– IntToStr(x) – функция, возвращающая строковое значение, содержащее целое число x,

– FloatToStr(x) – функция, возвращающая строковое значение, содержащее вещественное число x.

Заметим, что данные функции не будут работать, если вы программируете в среде Borland Pascal.

14.3. Примеры работы со строками

Пример 1. Подсчитать, сколько раз в строке встречается пробел.

Узнать, встречается ли, он в принципе можно, используя функцию Pos. Если она вернет 0, то пробел не встречается, иначе мы узнаем, где именно находится первый пробел, однако ничего не узнаем об остальных пробелах.

Для решения задачи переберем все символы строки и, используя переменную-счетчик, подсчитаем, сколько из них являются пробелами. Оформим решение в виде функции

```
function SpaceNumber(s: string): integer;
var
   i, n: integer;
begin
   n := 0;
   for i := 1 to length(s) do
      if s[i] = ' ' then
         n := n + 1;
   SpaceNumber := n;
end;
```

Пример 2. Стандартная функция Pos обнаруживает первое появление некоторой подстроки. Создадим функцию, которая возвращает n-е появление подстроки.

Детали работы функции описаны в программных комментариях:

```
function PosN(Substr, S, n): integer;
var
   i, pn, SubLen: integer;
begin
   SubLen := length(Substr);
   pn := 0; {в этой переменной считаем, сколько раз
                  уже встретили подстроку}
   i:=1;
   while (pn < n)and(i <= lenth(S) - SubLen + 1) do
   {Перебираем элементы строки, пока не найдем нужное
                  количество появлений подстроки
                  или не дойдем до конца строки}
```

```
    begin
       if Substr = Copy(S, i, SubLen) then
          pn := pn + 1;
       i := i + 1;
    end;
    if pn = n then
       PosN := i
    else
       PosN := 0; {если подстрока встречается меньше,
                     чем n раз, возвращаем 0}
end;
```

14.4. Текстовые файлы

Чтение и запись текстовых файлов производятся теми же процедурами readln и writeln, которые читают и записывают что-то на экране (по сути, Паскаль рассматривает экран как частный случай текстового файла). Чтобы осуществлять ввод из файла и вывод в файл, необходимо проделать следующие действия.

1) Описать так называемую файловую переменные типа text:

```
var
   f: text;
```

Если требуется одновременно работать с несколькими файлами, то для каждого необходима своя файловая переменная.

2) Каждую файловую переменную необходимо связать с конкретным файлом на диске с помощью процедуры Assign(<файловая переменная>, ‘<имя файла>’). Имя файла – это строковая константа (конкретное имя в кавычках) или переменная типа string, содержащая такое имя. Например:

```
Assign(f, 'c:\temp\sample.txt');
```

Если такого файла нет, то процедура его создаст.

3) Прежде чем что-либо читать из файла, его нужно открыть для чтения. Это делается процедурой Reset(<файловая переменная>). Например:

```
Reset(f);
```

4) После этого можно считывать данные из файла с помощью процедуры readln(<файловая переменная>, <переменная>). Например:

```
readln(f, s);
```

Первый после открытия для чтения readln прочитает первую строчку файла и перенесет ее содержимое в переменную s. Повторный вызов readln прочитает вторую строчку и т.д. Существует специальная переменная – файловый указатель, которая хранит номер текущей строки, т. е. определяет, какая строка будет считана очередным readln. При вызове Reset он устанавливается равным 0. Каждый вызов readln увеличивает его на 1.

Переменная s может иметь строковый или числовой тип. Однако если это числовая переменная, а в файле находятся нечисловые данные, то возникнет неисправимая логическая ошибка и программа аварийно завершится.

Если в программе стоит readln, а строчки в файле уже закончились, то также произойдет аварийное завершение программы. Чтобы этого избежать, существует функция EoF(<файловая переменная>) (EoF расшифровывается как End Of File), которая возвращает логическое значение true, если все строки из файла уже прочитаны, и false, если еще есть что читать.

Пример 1. Напечатаем содержимое файла sample.txt на экране.

```
var
   f: text;
   s: string;

begin
   {Связываем файловую переменную с конкретным фай-
       лом}
   Assign(f, 'sample.txt');
   {Открываем файл для чтения, указатель ставим в 0}
   Reset(f);
   {Повторяем, пока указатель не будет указывать на
       конец файла}
   while not EoF(f) do

   begin
      {Считываем одну строку в s, указатель увеличи-
       вается на 1}
      readln(f, s);
      {Печатаем строку s на экране}
      writeln(s);
   end;
   {Открытый для чтения файл следует закрыть}
   Close(f);
end.
```

Простого способа узнать, сколько в файле строк, не существует. Для этого следует считывать все строки до конца файла, как в предыдущем примере, и при этом увеличивать на 1 какую-нибудь переменную-счетчик.

Простого способа повторно прочитать некоторую строку также нет. Для этого следует снова вызвать Reset, обнулив таким образом указатель, а затем с помощью readln считать все предыдущие строки. Только после этого очередной readln прочитает нужную строку. По этой причине однажды считанные данные следует запоминать, например, в элементах массива.

5) Прежде чем что-либо записать в файл, следует открыть его для записи. Это можно сделать с помощью одной из двух процедур: Rewrite(f) или Append(f). Первая стирает содержимое файла (если оно там было) и устанавливает указатель на 0. Говорят, что файл открывается для перезаписи. Вторая открывает файл для дозаписи. Старое содержимое не стирается, указатель устанавливается в конец файла.

6) После того как файл открыт для записи, можно добавлять в него новые строчки с помощью процедуры writeln(<файловая переменная>, <число или строка>). Для примера запишем в файл sample.txt числа от 1 до 10:

```
var
   f: text;
   i: integer;

begin
   Assign(f, 'sample.txt');
   Rewrite(f);
   for i:=1 to 10 do
      writeln(f, i);
   Close(f);
end.
```

7) Однажды открытый для чтения или записи файл следует закрыть. Это делается процедурой Close(<файловая переменная>). Примеры ее использования уже встречались выше. Закрытие освобождает занятую под процессы чтения/записи память. Если производилась запись, то закрытие гарантирует, что все изменения будут сохранены на диске.

Задание 14. Символы, строки, текстовые файлы

Символьный тип

1) Попробуйте извлечь звук Beep путем печати соответствующего управляющего символа. PascalABC не бипает. Если под рукой нет Borland Pascal или Free Pascal или Delphi, то делайте следующее задание.

2) Напечатайте на экране кодовую таблицу в виде «символ – его код». Чтобы все символы уместились на экране, печатайте несколько столбцов (либо печатайте столько, сколько умещается, а затем пусть программа ждет нажатия Enter и выводит следующую порцию символов). Для разделения столбцов используйте символ табуляции (#9).

3) В Паскале имеется функция, делающая из строчной буквы прописную (UpCase), но нет обратной функции. Создайте ее. Грамотное с точки зрения английского языка название LowCase.

Замечание: В PascalABC есть LowCase, но вы все равно создайте свою процедуру.

Строки

1) Создайте программу, которая по трем введенным строкам, содержащим фамилию, имя и отчество, формирует и печатает одну строку, содержащую все вместе. Например, на ввод

Ivanov
Ivan
Ivanovich

она должна сформировать и напечатать строку ‘Ivanov Ivan Ivanovich’.

2) Создайте функцию, которая по строке, содержащей фамилию, имя и отчество, формирует строку, содержащую фамилию с инициалами. Например, по ‘Ivanov Ivan Ivanovich’ создаст ‘Ivanov I.I.’

3) Усовершенствуйте стандартную процедуру readln (создав ее аналог, который будет, например, называться readln2) таким образом, чтобы она:

а) позволяла вводить целое число,

б) если ввод вместо числа содержит что-то другое (например, буквы), информировала пользователя об ошибке ввода и запрашивала число снова до тех пор, пока не будет введено именно число.

Стандартная readln при неправильном вводе вызывает неисправимую ошибку, и программа аварийно завершается.

4) Создайте функцию, возвращающую заданное строкой арифметическое выражение. Например, получив строку ’2*2’, функция должна возвращать 4. Выражение может содержать числа и знаки операций (+, –, *, /) и не содержать скобок.

Текстовые файлы

1) Создайте процедуры для записи значений массива в текстовый файл и для считывания значений элементов массива из текстового файла.

2) Текстовый файл acc.txt (http://tvd-home.ru/docs/acc.txt), содержит временной ряд (в виде столбика цифр), представляющий собой запись ускорения движения руки при патологическом треморе (дрожание руки при болезни Паркинсона). При этом в качестве разделителя целой и дробной части используется запятая. Для обработки файла готовой программой требуются числа с точкой в качестве разделителя. Преобразуйте файл, заменив все запятые точками. Для этого, в частности, создайте процедуру или функцию, меняющую в строке все символы c1 на символы c2 (c1 и c2 должны быть параметрами).

3) Файл eeg.txt (http://tvd-home.ru/docs/eeg.txt) содержит запись ЭЭГ (электроэнцефалограммы) человека, включающую сигналы с 16 электродов (отведений). Каждому отведению соответствует столбец чисел. Столбцы разделены символом табуляции. Создайте процедуру, выделяющую в отдельный текстовый файл отведение с заданным номером.

15. Записи

15.1. Необходимость агрегации данных

Реальные объекты определяются набором взаимосвязанных характеристик. Скажем, вектор или точка задаются набором своих координат. Хранение таких характеристик в отдельных переменных провоцирует ошибки. Можно, например, взять часть координат от одной точки, а часть от другой или подвергнуть преобразованию не все координаты и т.д. По этой причине следует стремиться соединять (агрегировать) взаимосвязанные данные. Логическая связь между объектами решаемой задачи должна не только существовать в голове программиста, но и отражаться в используемых конструкциях языка программирования.

Простейшей конструкцией, позволяющей соединять однотипные данные, является массив. Для хранения координат точек (все координаты имеют один и тот же тип, например real) массивов вполне достаточно. Если все координаты хранятся в одной переменной типа массив, то они могут быть одновременно скопированы из одного места в другое, одновременно переданы какой-нибудь процедуре для обработки и т.п.

Однако если задача не абстрактно-математическая, то координаты могут быть интересны не сами по себе, а в связи с дополнительной информацией: координатами кого или чего они являются. Скажем, если имеются в виду координаты населенных пунктов, может потребоваться хранить также названия этих пунктов, которые представляют собой текстовые строки. По названию населенного пункта может потребоваться найти его координаты, или, введя координаты, вы захотите получить названия ближайших населенных пунктов. Массивы в такой ситуации не очень удобны. Придется создавать отдельно массив названий и массив координат.

Чтобы объединить в одну структуру (хранить в одной переменной) данные разных типов, в дополнение к массивам в Паскале предусмотрен еще один структурный тип – *запись*.

15.2. Тип-запись

Запись – это структура данных, состоящая из фиксированного числа компонентов, называемых *полями записи*. Компоненты (поля) записи могут быть различного типа, работать с ними можно так же, как с обычными переменными. Структура объявления типа-записи такова:

```
<имя типа> = record
  <список полей>
end;
```

Здесь <имя типа> – любой правильный идентификатор, <список полей> представляет собой последовательность разделов записи, между которыми ставится точка с запятой. Каждый раздел состоит из одного или несколь-

ких идентификаторов полей, за которыми ставится двоеточие и описание типа поля (полей). Например:

```
{Запись, хранящая название и координаты населенного пункта}
TSettlements = record
   x, y: real; {координаты}
   Name: string; {название}
end;
```

Данная запись содержит три поля (x, y, Name) в двух разделах. Поля записи могут быть любого типа, кроме файлового.

После того как тип-запись описан, можем вводить его переменные:

```
var
   x, z, s: TSettlement;
```

Единственная операция, которую можно производить непосредственно с переменными типа запись, это присваивание. Иными словами, содержимое одной записи копируется в другую оператором

```
x := s;
```

С отдельными полями записи можно работать как с обычными переменными. Для них разрешены все операции, допустимые для типа этого поля. Для доступа к полям записи используется составное имя – сначала идет имя записи, затем точка и имя соответствующего поля, например:

```
s.x := 64;
s.y := 32;
s.Name := 'Saratov';
x.x := s.x + 20;
```

15.3. Оператор присоединения *with*

Чтобы сделать обращение к полям записи более коротким, используется оператор присоединения **with**:

```
with <переменная> do
begin
   <операторы>
end;
```

<переменная> – имя переменной типа запись,
<операторы> – любые операторы Паскаля.

Внутри блока операторов можно при обращении к полям записи опускать имя переменной. Вместо неуказанного имени по умолчанию будет добавлено имя из первой строчки оператора. Например:

```
with s do
begin
```

```
   x := 64; {сокращенное обращение к полю x записи s,
                    то же что s.x}
   y := 32; {сокращенное обращение к полю s.y}
   Name := 'Saratov';
   z.y := y+20; {можно обращаться к полям других за-
                    писей, но придется использовать
                    полную форму}
{А вот так делать нельзя}
   x.y := y+20; {обращаться  к полям  записи с именем
                 <em>x</em>  внутри  такого оператора
                 нельзя из-за совпадения  имен записи
                 x и одного из полей записи s}
end;
```

15.4. Примеры использования записей

Пример 1. Использование типа запись для хранения результатов экзамена, содержащего название предмета и оценку:

```
type
   TExamRes = record
      Subject: string;
      Mark: 2..5;
   end;
```

Большая буква T в начале имени типа TExamRes добавлена из соображений стиля. Принято добавлять букву T ко всем идентификаторам вводимых вами типов.

Пример 2. Использование типа запись для хранения результатов сессии, содержащего имя студента и результаты экзаменов:

```
TSession = record
   Name: string;
   ExamResults: array [1..5] of TExamRes;
end;
```

Здесь одним из полей является целый массив элементов определенного выше типа TExamRes. Соответственно если есть переменная этого типа

```
var
   s: TSession;
```

тогда s.Name – имя студентов, s.ExamResults[3].Subject – название 3-го экзамена и т.д.

Пример 3. Тип для хранения результатов сессии студенческой группы:

```
TGroupResults = array [1..10] of TSession;
```

Элементы массива могут иметь любой тип, в том числе и тип запись. Чтобы совместно с результатами хранить номер группы, можно использовать не тип массив, а новую запись:

```
TRecGroupResults = record
   Number: integer; {Номер группы}
   GroupResults: array [1..10] of TSession;
end;
```

Если есть переменная

```
var
   Res: TRecGroupResults;
```

то

`Res.Number` – номер группы;

`Res.GroupResults[i]` – результаты *i*-го студента;

`Res.GroupResults[i].Name` – имя *i*-го студента;

`Res.GroupResults[i].ExamResults[k].Mark` – оценка *i*-го студента за *k*-й экзамен.

Пример 4. В текстовом файле students.txt хранится информация о студентах: фамилия, имя, отчество, год рождения и какую школу окончил. Примеры строк из этого файла:

Иванов Иван Иванович 1985 11
Петров Петр Петрович 1986 102

Требуется описать массив записей с информацией о студентах и прочитать данные из файла в этот массив. Программа также должна уметь по введенному номеру школы выдавать список закончивших ее студентов:

```
program Students;

type
   {Тип запись с информацией про студента}
   TStudent = record
      F, I, O: string;
      Year: 1900..2100;
      School: 1..200;
   end;

var
   {Массив информации о студентах}
   Students: array [1..1000] of TStudent;
   SNum: integer; {количество студентов}
   SN: 1..200; {номер школы}
   i: integer;
```

```
procedure ParseString(s: string; var F, I, O: string;
var Year, School: integer);
{Процедура, которая из строки, считанной из файла,
выделяет фамилию, имя, отчество, год рождения и номер
школы}
var
   n, code: integer;
begin
   {Положение 1-го пробела в строке}
   n:=Pos(' ', S);
   {Выделение фамилии}
   F:=Copy(S, 1, n-1);
   {Берем все, что идет после 1-го пробела}
   S:=Copy(S, n+1, length(S));
   {Положение 1-го пробела в строке без фамилии}
   n:=Pos(' ', S);
   {Выделение имени}
   I:=Copy(S, 1, n-1);
   {Берем все, что идет после 1-го пробела}
   S:=Copy(S, n+1, length(S));
   n:=Pos(' ', S);
   O:=Copy(S, 1, n-1);
   S:=Copy(S, n+1, length(S));
   n:=Pos(' ', S);
   Val(Copy(S, 1, n-1), Year, code);
   //Year:=StrToInt(Copy(S, 1, n-1)); - для PascaABC
   S:=Copy(S, n+1, length(S));
   n:=Pos(' ', S);
   Val(Copy(S, 1, n-1), School, code);
   //Scool:=StrToInt(Copy(S, 1, n-1));
end;

procedure ReadData(FileName: string;
   var Students: array of TStudent; var Num: inte-
ger);
{Процедура, читающая данные из файла и определяющая
их общее количество}
var
   F: text;
   S: string;
begin
   AssignFile(F, FileName);
   Reset(F);
   Num:=0;
   while not EOF(F) do
```

```
    begin
       Readln(F, S);
       with Students[Num] do
          ParseString(S, F, I, O, Year, School);
          Num:=Num+1;
    end;
    CloseFile(F);
end;

begin
    ReadData('students.txt', Students, SNum);
    Write('Input school number');
    Readln(sn);
    {Поиск студентов из школы sn}
    for i:=1 to SNum do
       with Students[i] do

       begin
          if School = sn then
             writeln(F, ' ', I, ' ', O);
       end;

    readln;
end.
```

Задание 15. Записи

1. Время суток задается в формате чч:мм:сс. Создайте процедуру, которая по заданному времени вычисляет, какое время получится через одну минуту. Для хранения информации о времени используйте тип запись.

2. Создайте процедуру, которая будет печатать запись из предыдущей задачи в формате чч:мм:сс. Скажем, если поля записи равны 12 (часы), 5 (минуты) и 0 (секунды), процедура должна сформировать и напечатать строку 12:05:00.

3. Определите тип данных, хранящий информацию о прямой. Создайте процедуру, вычисляющую прямую, проходящую через 2 точки.

4. Создайте текстовый файл, хранящий информацию из багажной ведомости камеры хранения, включив следующую информацию: фамилию, имя, отчество пассажира, количество и общий вес вещей. Выведите в новый файл записи о пассажирах, суммарный вес вещей которых больше 10 кг.

5. Составьте список студентов, включающий фамилию, имя, отчество и 5 оценок. Напишите программу, удаляющую из списка тех, кто имеет хотя бы одну двойку.

16. Указатели

16.1. Ссылочные типы и указатели

Каждая область памяти имеет свой адрес. Грубо говоря, все байты пронумерованы, эти номера и есть адреса. Значения переменных хранятся в памяти, и соответственно у каждой переменной есть свой адрес. Указателем называется переменная, в которую записан адрес некоторой области памяти. Например, в указателе может быть записан адрес, по которому хранится значение какой-нибудь другой переменной.

Типы переменных для хранения указателей называются ссылочными типами. Для описания ссылочного типа используются символ «^» и имя базового типа. Базовым типом может быть любой тип, кроме файлового, включая описанные вами самими типы.

Примеры описания ссылочных типов:

```
type
   PInteger = ^integer;
   PReal = ^real;

   {Обычный тип-массив, не ссылочный}
   TMyArray = array [1..10] of real;
   {А это уже соответствующий ссылочный тип}
   PMyArray = ^TMyArray;
```

Значения с такими типами будут хранить адреса, по которым в памяти могут располагаться соответственно целое число, вещественное число и массив вещественных чисел.

Для того чтобы присвоить указателю некоторое значение, можно воспользоваться операцией взятия адреса, которая обозначается символом @. Пусть, например, описаны переменные

```
var
   X, Y: integer; {обычные переменные, не указатели}
   P: ^integer; {указатель на целое число}
   P2: PInteger; {тоже указатель на целое число, тип
которого был описан выше}
```

Тогда допустима следующая операция:

```
P := @X;
```

Значением указателя P станет адрес, по которому хранится значение переменной X.

Зная адрес, можно определить, какое значение по этому адресу записано, или записать значение в память с этим адресом. Для этого используется так называемая операция *разыменования* указателя (dereference, если по-английски). Она обозначается символом «^», который ставится после имени переменной-указателя. Например:

```
{В область памяти с адресом P записывается число 5}
P^ := 5;
{В переменную X записывается значение, хранящее по адресу P}
X := P^;
```

С разыменованными указателями (после которых поставили символ «^») можно работать как с обычными переменными.

Пусть выполняются операторы:

```
X := 3;
P := @X;
P^ := 5;
```

В результате в переменную X будет записано число 5. Мы это сделали косвенным способом, изменив содержимое ячеек памяти, где хранилось значение переменной X.

Ссылочные типы допустимо образовывать от любого типа, в том числе от другого ссылочного типа. Поэтому допустимо определение указателя на указатель:

```
var
   z: ^PInteger;
```

Возможные значения такой переменной – адреса в памяти, где хранятся адреса, по которым записаны целые числа.

Среди возможных значений указателей выделяется одно особенное значение – **nil**. Указатель, имеющий значение **nil**, «никуда не указывает». Указатель **nil** считается константой, которую можно присвоить указателю любого типа.

16.2. Нетипизированные указатели

Хотя значениями ссылочных типов, образованных от любого базового типа, являются адреса, в памяти прямое присваивание значений разнотипных указателей запрещено. Пусть, например, имеется описание

```
var
   x, y: ^integer;
   z: ^real;
```

Допустимы присваивания однотипных указателей

```
x := y;
```

Недопустимы присваивания

```
x := z;
z := y;
```

Такое ограничение можно обойти. Для этого существуют указатели, не связанные с определенным типом данных, имеющие тип Pointer:

```
var
   p: Pointer;
```

Нетипизированные указатели совместимы с любыми типизированными укзателями. Соответственно допустимо

```
p:=x;
z:=p;
```

16.3. Динамическое выделение памяти

Память под обычные переменные выделяется одновременно с началом работы программы (или подпрограммы для локальных переменных) и освобождается по окончании работы программы (подпрограммы). Такие переменные называются *статическими*.

Использование указателей позволяет создавать и уничтожать переменные в любом месте программы. Под образованием и уничтожением переменных имеется в виду отведение памяти, достаточной для хранения значения базового для указателя типа, и соответственно ее освобождение.

Без такого *динамического* выделения памяти невозможно представить себе работу никакой сложной программы. Предположим, например, что ваша операционная система при запуске резервировала память под все возможные задачи, которые она умеет выполнять. При работе с большими структурами данных почти всегда используются указатели.

Выделение и освобождение памяти в Паскале делается стандартными процедурами New и Dispose. Пусть p – типизированный указатель.

New(p) выделяет память под хранение значения базового для указателя p типа и записывает в адрес этой области памяти.

Dispose(p) освобождает память, на которую ссылается указатель p. Значение указателя при этом не меняется, но незарезрвированную память могут использовать другие приложения, так что работать с ней во избежание конфликтов с другими программами не рекомендуется. Хорошим тоном будет записать в переменную p значение **nil** сразу после освобождения памяти процедурой Dispose.

Для нетипизированных указателей эти процедуры неприменимы, так как непонятно, сколько байт памяти надо выделять. Для резервирования и возвращения памяти в этом случае используются процедуры GetMem и FreeMem.

GetMem(p, size) выделяет size байт и записывает их адрес в нетипизированный указатель p.

FreeMem(p, size) освобождает память, имеющую адрес p. size, – размер ранее зарезервированной памяти.

Всю доступную оперативную память компьютера принято называть *кучей*. При динамическом выделении памяти говорят о взятии ее из кучи. При освобождении памяти (Dispose или FreeMem) говорят, что она возвращена в кучу.

16.4. Рекурсивные структуры данных

Пусть описан тип запись, и одним из полей этой записи является указатель. В этот указатель можно записать адрес, по которому данная запись располагается, либо, что более интересно, адрес другой записи того же типа. Это позволит при помощи указателей создать структуру данных, называемую *связанным списком*.

Пример описания такой структуры данных:

```
type
   PListElement = ^TListElement;
   TListElement = record
      <Произвольные поля записи>
      NextElement: PListElement;
   end;
```

Здесь сначала описывается тип – указатель PListElement – на запись типа TListElement. Затем идет описание самого типа записи, одно из полей которой имеет тип PListElement. Получается, что мы используем идентификатор TListElement еще до того, как его описали. Обычно такие вещи запрещено делать, но в данном случае, когда описывается ссылочный тип, из этого правила сделано исключение.

Если имеется запись этого типа, то можно с помощью указателя динамически создать еще один элемент такого же типа. По указателю в этом новом элементе – еще один элемент и т.д., сколько потребуется. Такая структура данных называется однонаправленным связанным списком. Ее схематическое изображение показано на рис. 16.1.

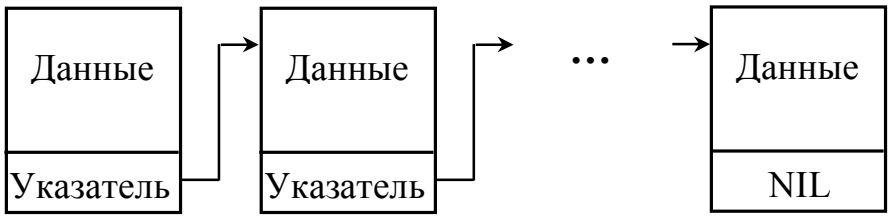


Рис. 16.1

Задача 1. Пусть в виде однонаправленного связанного списка хранятся целые числа. Создать процедуры, которые:

а) создают список, динамически выделяя из кучи память под заданное число элементов N;

б) заполняют информационную часть элементов списка числами 1, 2, …, N;

в) печатают в столбик информационную часть списка;

г) уничтожают список, возвращая в кучу выделенную под него память.

Решение:

```
type
   PList = ^TList;
```

```
   TList = record
     x: integer; {информационное поле}
     Next: PList;{ссылка на следующий элемент списка}
   end;
procedure CreateList(var Head: PList; Num: integer);
{Процедура создает список, содержащий Len элементов,
      начинающийся с элемента Head}
var
   n: integer;
   H: PList;
begin
   if Num > 1 then
   begin
      New(Head);
      H := Head;
      {При работе с однонаправленными списками прин-
      ципиально важно не потерять ссылку на первый
      элемент. Поэтому дальнейшие операции будем про-
      водить не с Head, а со вспомогательной перемен-
      ной H}
      for n := 2 to Num do
      begin
         New(H^.Next);
         H := H^.Next;
      end;
      H^.Next := nil;
   end else
      Head := nil;
end;
procedure FreeList(var Head: PList);
{Процедура, освобождающая память, выделенную под спи-
      сок с первым элементом Head}
var
   H: PList;
begin
   {Поскольку список все равно уничтожается, можем
      спокойно изменять Head}
   while Head <> nil do
   begin
      H := Head;
      Head := Head^.Next;
      Dispose(H);
   end;
   Head := nil;
end;
```

```
procedure FillList(Head: PList);
{Процедура заполняет информационную часть элементов
      списка числами 1, 2, 3, …}
var
   n: integer;
begin
   n:=1;
   while Head <> nil do
   begin
      Head^.x := n;
      n := n+1;
      Head:=Head^.Next;
   end;
end;

procedure PrintList(Head: PList);
{Процедура печатает информационную часть элементов
      списка}
begin
   while Head <> nil do
   begin
      writeln(Head^.x);
      Head := Head^.Next;
   end;
end;
```

Программа, работающая с этими процедурами, может выглядеть, например, так:

```
var
   List: PList;
begin
   CreateList(List, 5); {создаем список из 5 элемен-
                         тов}
   FillList(List); {в информационные части пишем чис-
                    ла 1, 2, 3, 4, 5}
   PrintList(List);  {печатаем информационные части
                    элементов списка}
   ClearList(List); {освобождаем память}
   readln;
end.
```

Связанные списки, так же как и массивы, позволяют хранить набор однотипных данных и могут рассматриваться как альтернатива массивам. Какую структуру предпочесть, массивы или связанные списки, зависит от тех операций, которые чаще всего будут производиться с данными.

Например, у связанных списков легче производить операцию вставки и удаления элементов внутри структуры. У массива надо смещать все последующие элементы, в списках достаточно поменять указатель одного только предыдущего элемента.

В то же время, если надо получить доступ к элементу по его номеру, в массиве это делается сразу, а в списке придется пройти все цепочку, начиная с головного элемента.

В рассмотренных однонаправленных списках, зная элемент, можно было найти все последующие элементы. Однако если известна ссылка на один из средних элементов, то идущие перед ним найти не удастся. Если в нахождении предыдущих элементов есть нужда, то создаются двунаправленные списки, каждый элемент которых содержит два указателя – на предыдущий и последующий элементы.

Вообще говоря, каждый элемент подобной структуры может содержать сколько угодно указателей на элементы того же типа, что позволяет формировать не только списки, но и деревья и графы.

Задание 16. Указатели

1. Присвойте какой-нибудь переменной значение не напрямую, а узнав ее адрес и изменив содержимое памяти по этому адресу.

2. Узнайте, что станет с переменной типа char, если в соответствующую ей память записать целое число.

3. Для однонаправленного списка из целых чисел напишите следующие процедуры:

а) создать список и заполненить его случайными целыми числами. В лекции приведен пример создания списка, начиная с 1-го элемента, попробуйте это сделать, начиная с последнего;

б) напечатать значения элементов списка;

в) исключить из списка элемент с заданным номером;

г) поменять местами два элемента списка;

д) уничтожить список.

4. В текстовом файле содержится столбик чисел. Создайте новый файл, где будут те же числа, но записанные в обратном порядке. Для промежуточного хранения данных используйте однонаправленный список.

5. Создайте однонаправленный список из целых чисел. Перестройте элементы списка в обратном порядке.

6. По строке, содержащей арифметическое выражение, куда входят числа и символы операций (+, -, /, *), постройте бинарное дерево и создайте функцию, которая по такому дереву вычисляет значение выражения.

Список рекомендуемой литературы

Общие вопросы программирования

Ахо, А. В. Структуры данных и алгоритмы / А. В. Ахо, Дж. Хопкрофт, Дж. Д. Ульман : пер. с англ. – М. : Изд. дом «Вильямс», 2003. – 384 с.

Вирт, Н. Алгоритмы и структуры данных. Новая версия для Оберона / пер. с англ. Ф. В. Ткачёва. – М. : ДМК Пресс, 2010. – 272 с.

Кнут, Д. Искусство программирования. Т. 1. Основные алгоритмы / Д. Кнут. – 3-е изд. – М. : Изд. дом «Вильямс», 2006. – 720 с.

Круз, Р. Л. Структуры данных и проектирование программ : учеб. пособие / Р. Л. Круз; пер. 3-го англ. изд. К. Г. Финогенова. – М. : Бином. Лаб. знаний, 2008. – 765 с.

Окулов, С. М. Программирование в алгоритмах : учеб. пособие / С. М. Окулов. – М. : Бином. Лаб. знаний, 2004. – 341 с.

Себеста, Р. У. Основные концепции языков программирования / Р. У. Себеста. – 5-е изд. : пер. с англ. – М. : Изд. дом «Вильямс», 2001. – 672 с.

Синицын, С. В. Программирование на языке высокого уровня: учебник / С. В. Синицын, А. С. Михайлов, О. И. Хлытчиев. – М. : Изд. центр «Академия», 2010. – 302 с.

Изучение языка Паскаль

Алексеев, Е. Р. Free Pascal и Lazarus : учебник по программированию / Е. Р. Алексеев, О. В. Чеснокова, Т. В. Кучер. – М. : AltLinux, 2010. – 438 с.

Андреева, Т. А. Программирование на языке Pascal : учеб. пособие / Т. А. Андреева. – М. : Интернет-Ун-т Инф. Технологий : Бином. Лаб. знаний, 2010. – 234 с.

Григорьев, С. А. Программирование на языке Паскаль для математиков : учеб. пособие / С. А. Григорьев. – Калининград : Изд-во Калининград. ун-та, 1997. – 92 с.

Епанешников, А. М. Программирование в среде Turbo Pascal 7.0 : учеб. пособие / А. М. Епанешников, В. А. Епанешников. – 3-е изд. – М. : Диалог-Мифи, 1995. – 368 с.

Зуев, Е. А. Программирование на языке Turbo Pascal 6.0, 7.0 / Е. А. Зуев. – М. : Радио и связь : Веста, 1993. – 380 с.

Марченко, А. И. Программирование в среде Turbo Pascal 7.0 / А. И. Марченко, Л. А. Марченко. – 5-е изд., доп. и перераб. – Киев : ВЕК+, 1999. – 458 с.

Огнёва, М. В. Turbo Pascal : первые шаги. Примеры и упражнения : учеб. пособие / М. В. Огнёва, Е. В. Кудрина. – 3-е изд., доп. и перераб. – Саратов : Науч. кн., 2008. – 99 с.

Семашко, Г. Л. Программирование на языке Паскаль / Г. Л. Семашко, А. И. Салтыков. – 2-е изд., перераб. и доп. – М. : Наука, 1993. – 207 с.

Оглавление

Учебное издание

Диканев Тарас Викторович,
Вениг Сергей Борисович,
Сысоев Илья Вячеславович

ПРИНЦИПЫ И АЛГОРИТМЫ ПРИКЛАДНОГО ПРОГРАММИРОВАНИЯ

Учебное пособие для студентов, обучающихся на факультете нано- и биомедицинских технологий

Редактор *Е. А. Малютина*
Технический редактор *В. В. Володина*
Корректор *А. Л. Шибанова*
Оригинал-макет подготовлен *О. Л. Багаевой*

Подписано в печать 15.10.2012. Формат 60x84 1/16.
Усл. печ. л. 8,14(8,75). Тираж 100. Заказ 66.

Издательство Саратовского университета.
410012, Саратов, Астраханская, 83.
Типография Издательства Саратовского университета.
410012, Саратов, Астраханская, 83.
139