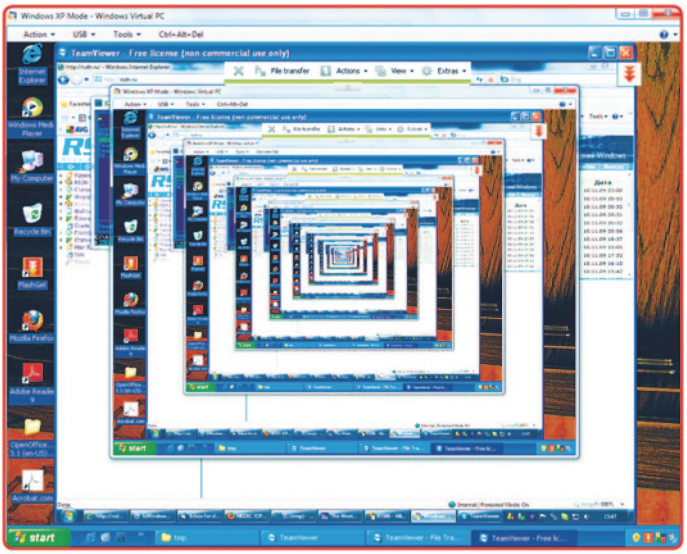
РАЗВИТИЕ ИНТЕЛЛЕКТА ШКОЛЬНИКОВ



С. М. Окулов

ОСНОВЫ ПРОГРАММИРОВАНИЯ

Лаборатория ЗНАНИЙ

РАЗВИТИЕ ИНТЕЛЛЕКТА ШКОЛЬНИКОВ

С. М. Окулов

ОСНОВЫ ПРОГРАММИРОВАНИЯ

10-е издание, электронное

Москва
Лаборатория знаний
2020

УДК 519.85(023)
ББК 22.18
О-52

Серия основана в 2008 г.

Окулов С. М.

О-52 Основы программирования / С. М. Окулов. — 10-е изд., электрон. — М. : Лаборатория знаний, 2020. — 339 с. — (Развитие интеллекта школьников). — Систем. требования: Adobe Reader XI ; экран 10". — Загл. с титул. экрана. — Текст : электронный.

ISBN 978-5-00101-759-2

В книге рассмотрены фундаментальные положения программирования: конечная величина и конструируемые на ее основе различные типы данных; управляющие конструкции — элементарные составляющие любого алгоритма и основа управления вычислительным процессом; структуризация задач как основополагающий механизм их реализации на компьютере; упорядочение (сортировка) как основа эффективной работы с любыми данными и, наконец, перебор вариантов, как универсальная схема компьютерного решения задач.

Для учащихся старших классов, студентов и учителей информатики.

УДК 519.85(023)
ББК 22.18

Деривативное издание на основе печатного аналога: Основы программирования / С. М. Окулов. — 9-е изд. — М. : Лаборатория знаний, 2018. — 336 с. : ил. — (Развитие интеллекта школьников). — ISBN 978-5-00101-136-1.

ISBN 978-5-00101-759-2

Содержание

Предисловие

Данный учебник является переработанным вариантом книги автора «Основы программирования», которая выходит в издательстве «БИНОМ. Лаборатория знаний» начиная с 2002 года (пять изданий). Но даже не этот момент времени следует считать точкой отсчета. Приведем фрагмент предисловия к книге «Основы программирования».

«Из истории возникновения учебника. В 1988 году перед автором возникла проблема: «чему учить в информатике и как учить». Исходным «багажом» при принятии решения было образование в классическом университете по специальности «прикладная математика» и более чем 15-летний опыт разработки программного обеспечения специализированных вычислительных комплексов в промышленности. Попытки обучения по существующим в то время учебникам, а их было не так много, как в настоящее время, не принесли ни удовлетворения, ни результатов. Информатика воспринималась как обычный предмет. Дидактические возможности компьютера не использовались в полной мере. Может быть, причиной явилось отсутствие у автора соответствующего опыта и педагогического образования. Ограничимся констатацией факта, оставим критику этого результата, так же как и обсуждение достоинств и недостатков существующих учебников, доброжелателям.

Тезисно обозначим исходные положения при принятии решения.

Первой посылкой было убеждение в том, что программирование является «стержнем информатики». В нем синтезировано все, что десятилетиями нарабатывалось в Computer Science. Это и результаты работы специалистов, работающих на «стыке» математики и информатики, это и достижения в вычислительной технике, это и, наконец, огромный опыт формализации и решения сложнейших проблем в самом программировании, связанный с созданием больших программных комплексов.

Второй исходной посылкой является убеждение в том, что занятия по информатике в корне должны отличаться от традиционных занятий по любому другому предмету. Во-первых, на занятиях по информатике должна поощряться ошибка, ибо только через ошибку можно прийти к результату, при изучении же любого другого предмета ошибка карается двойкой. Во-вторых, постоянная обратная связь с обучаемым через компьютер, объективная и лишенная эмоций, — это инструментарий индивидуального и развивающего обучения. В-третьих, стиль мышления у программистов свой, отличающийся от стиля мышления как математика, так и любого другого специалиста. Он настроен, если так можно выразиться, на борьбу с хаосом. Любая сложная программа — это миллионы и более составляющих, движущихся и взаимодействующих. И в результате этого взаимодействия должен получаться определенный результат. Представим себе техническую систему такого уровня сложности...

Итак, за основу обучения следует взять программирование, с максимальным использованием компьютера на занятиях, и при этом должен формироваться определенный стиль мышления. В таком ключе и шла вся последующая работа. Большое влияние на нее оказала подготовка школьников к олимпиадам по информатике. Синтезировались и обобщались все педагогические находки, которые затем находили применение в преподавании информатики и подтверждали обозначенные выше положения.

Основной методический принцип учебника — все познается через труд, через преодоление ошибок (собственных), через процесс решения задач. Этот принцип определяет структуру занятий. Вводная часть – обсуждение нового материала, эксперименты с заготовками решений задач, самостоятельное решение задач.

В идейном плане наиболее близкими к данному учебнику, несмотря на кажущиеся принципиальные отличия, являются учебники А. Г. Кушниренко и Г. В. Лебедева[1]. Эти учебники — наиболее цельные из существующих по идеям, заложенным в них, и, следовательно, по содержанию. Они имеют свое «лицо» и идут к методике преподавания информатики от методики обучения математике. Единственное отличие, на наш

[1] Ныне незаслуженно забытые.

взгляд, заключается в том, что при совпадении идейных установок мы в методике преподавания идем от компьютера, решая проблемы обучения через практическое программирование. При этом компьютер, система программирования являются не самоцелью обучения, а инструментом для реализации целей, хотя при этом познается и сам инструментарий».

И в 2010 году автор ничего не хочет изменять из написанного. Более того, в развитие данного подхода за прошедшие годы, автор пытался ответить на вопрос, что является фундаментальным в информатике, какие понятия являются основополагающими. Если, например, в математике они известны — число и форма, в физике, химии и других областях знания они также определены, то что с информатикой? Этот вопрос — далеко не праздный. Ответ на него определяет многое, в частности судьбу школьного предмета информатики[2]. Точка зрения автора в развернутом виде изложена в его монографии[3], здесь же тезисно обозначим основные положения, кратко изложив один из ее параграфов.

Мы говорим о содержании курса информатики. Фундаментальных понятий должно быть немного, они должны пронизывать все содержание и на каждом витке содержания, образно выражаясь, иметь свою «окраску», свой уровень сложности.

Перечислим эти понятия.

1. Величины, структуры данных (в первую очередь массив). Структуры данных и величины — это тот инструмент, с помощью которого данные о проблеме оформляются таким образом, чтобы она могла быть воспринята и обработана компьютером.

2. Управление вычислительным процессом (управляющие конструкции, рекурсия).

3. Структуризация проблем (процедуры, функции — инструмент реализации принципа «разделяй и властвуй», механизмов абстрагирования, декомпозиции и формализации).

---

[2] Понятие «информатика» — достаточно объемное, включающее и кибернетику, и моделирование, и т. д. В данной книге мы, скорее всего, говорим о Computer Science, об отрасли человеческого знания, охватывающей решения проблем с использованием компьютера, но называем ее, в силу сложившейся традиции отождествления понятий при переводе, информатикой.

[3] Окулов С. М. Информатика: развитие интеллекта школьника. 2-е изд. – М.: БИНОМ. Лаборатория знаний, 2008.

4. Отношение порядка (упорядоченности) на множестве объектов определенной структуры.

5. Перебор вариантов в пространстве состояний задачи.

Итак, только пять, и выбор их не случаен. Коротко обоснуем его.

1. **Структуризация данных.** Фундаментальная основа любого управления есть данные (информация) и действия, совершаемые над данными (с информацией). При их структуризации очевидна аналогия со структурой ЭВМ (с ее процессором и памятью, организованной по принципу массива).

2. **Управляющие конструкции.** Перечень управляющих конструкций полный: любое алгоритмическое действие, вероятно, в любой области деятельности, может быть представлено с использованием только следования, ветвления и повторения. В классической работе[4] показано, что этих конструкций достаточно для реализации любого алгоритма.

Структуры данных и управляющие конструкции позволяют в компактной форме записать большое количество действий и обозначить большие объемы информации.

3. **Структуризация задач.** Нельзя объять необъятное. Человек при решении сложной проблемы всегда пытается выделить главное, расчленить ее на более простые. Такова природа человеческого интеллекта. По данным когнитивных психологов, человек может следить не более чем за семью непрерывно меняющимися во времени величинами, эффективно работать не более чем с пятью–семью людьми. Инструментами этого разъятия (разделения и властвования) являются процедуры и функции. Причем этот процесс — нелинейный. Если на каком-то этапе дальнейшее продвижение невозможно (из разъятого не удается методами синтеза создать гармонию), то человек возвращается в своих действиях назад и повторяет на новом витке, в новом варианте последовательный анализ проблемы.

4. **Отношение порядка** (упорядоченности) на множестве объектов определенной структуры. Множество ячеек оперативного запоминающего устройства упорядочено, множество регистров процессора имеют свои имена, множество дорожек магнитного диска упорядочено, множество электронных адре-

[4] Bohm C., Jacopini G. Flow Diagrams Turing Machines, and Languages with Only Two Formulation Rules//Communicatins of the ASM. 1966. May.

сов в сети Интернет связано отношениями порядка и т. д. Это один аспект — связь с нижним уровнем компьютера. С другой стороны, упорядоченность воспринимается как что-то естественное и для человека, на что даже не стоит специально обращать внимание. Действительно, данное понятие формируется на стадии конкретных операций (Ж. Пиаже), а это возраст от 7 до 11 лет.

Проблема упорядоченности неразрывно связана с другой проблемой — поиска данных (информации). Упорядоченность упрощенно можно трактовать как необходимое условие быстрого поиска. Проблема эта как была, так и остается одной из ключевых в информатике. Например, поиск общей подпоследовательности максимальной длины в двух текстах. Несмотря на возросшую многократно производительность компьютера, исследованием методов нахождения данных за возможно меньшее время по-прежнему занимаются ведущие ученые отрасли. Достижение, исследование упорядоченности на множестве объектов — очень непростая задача, даже на простых структурах данных. Фундаментальность понятия «упорядоченность» следует из того, что всякое развитие есть, в определенной степени, увеличение упорядоченности в системе.

5. **Перебор вариантов** в пространстве состояний задачи. В принципе универсальная, хотя далеко не всегда реально достижимая схема компьютерного решения задач. Особенность компьютера заключается в том, что он действует только с упорядоченными структурами данных. Только там, где есть отношение порядка, функционируют циклические конструкции, и т. д. — решаются переборные задачи. Только перечисляемое подвластно компьютеру, а в этом и есть сущность перебора вариантов.

Именно эти фундаментальные для информатики понятия и рассматриваются в данной книге. Двадцатипятилетний опыт работы в образовательной информатике позволяет утверждать, что школьник, своевременно их освоивший, может стать (при его желании) успешным специалистом в Computer Science.

Часть I

Программирование в среде Паскаль

1.1. Основные управляющие конструкции

Занятие № 1. Первая программа

План занятия

1. Краткое знакомство со средой программирования.
2. Экспериментальный раздел занятия.
3. Выполнение самостоятельной работы.

Краткое знакомство со средой программирования

После загрузки системы программирования на экране появляются три окна (рис. 1.1):

`File Edit ...`	– окно 1 – главное меню
`Line 1 Col 1 ...`	– окно 2 – основное или рабочее окно
`F1– Help ...`	– окно 3 – окно помощи, в нем указано назначение основных функциональных клавиш

Рис. 1.1. Схематическое изображение структуры экрана

Переход из первого окна во второе и наоборот осуществляется нажатием клавиши F10.

В рабочем окне редактора среды программирования наберем текст первой программы — вычисления произведения двух целых чисел:

```
Program My1_1;
 Var a,b,rez: Integer;
 Begin
  WriteLn('Введите два числа через пробел');
  ReadLn(a,b);
```

```
  rez:=a*b;
  WriteLn('Их произведение равно ',rez);
  WriteLn('Нажмите Enter');
  ReadLn;
End.
```

Программа начинается с **заголовка**, имеющего следующий вид:

```
Program  <имя программы>;
```

Имя нашей программы — My1_1. Заметим, что в имени программы не должно быть пробелов, оно должно начинаться с буквы, состоять только из латинских букв, цифр и некоторых символов, не допускается использование символов точки и запятой.

За заголовком идет **раздел описаний**, в котором должны быть описаны все идентификаторы (константы, переменные, типы, процедуры, функции, метки), которые будут использованы в программе. В данном случае из разделов описаний имеется лишь один — раздел переменных. Он начинается со служебного слова Var, после которого идет последовательность объявления переменных, разделенных точкой с запятой. В каждом объявлении перечисляются через запятую имена переменных (идентификаторы) одного типа, после чего ставится двоеточие и указывается тип переменных. В нашем примере описаны три переменные: *a*, *b* и *rez*; все они имеют целый тип (Integer), т. е. значениями переменных этого типа являются целые числа.

Понятие переменной — центральное в любом языке программирования. Для описания переменной (величины, которая изменяется в процессе работы программы) следует указать имя переменной, ее тип и значение. Соблюдается следующий принцип: использовать переменную можно лишь тогда, когда ей присвоено некоторое значение. Использование в выражениях неинициализированных переменных, то есть переменных, значения которым не были присвоены явно, часто является причиной ошибок.

После раздела описаний идет **раздел операторов**, который начинается со служебного слова Begin и заканчивается служебным словом End. В этом разделе задаются действия над объектами программы, введенными в разделе описаний. Операторы в этом разделе отделяются друг от друга точкой с запятой. После

последнего слова End ставится точка. После слова Begin ни точка, ни точка с запятой не ставятся.

Первый встречающийся оператор — это **WriteLn**('текст'); — записать (вывести) на экран текст, заключенный между апострофами и взятый в скобки. Ln добавляется в конце имени оператора для того, чтобы после вывода на экран текстов или результатов выполнения программы курсор автоматически переходил на следующую строку.

Следующий оператор — это **ReadLn**(a,b); — читать данные с клавиатуры. В данном случае необходимо ввести два целых числа через пробел, тогда переменной *a* присваивается значение, равное первому введенному числу, а переменной *b* присваивается значение, равное второму введенному числу. Например, вы ввели числа 12 и 45, тогда *a*=12, а *b*=45. В конце этого оператора также можно ставить Ln.

После этих двух операторов стоит оператор присвоения: rez:=a*b; (:= — это знак оператора присвоения в языке Паскаль; * — это знак умножения). Значение выражения из правой части оператора присвоения заменяет текущее значение переменной из левой части. Тип значения выражения должен совпадать с типом переменной. При выполнении оператора пе-

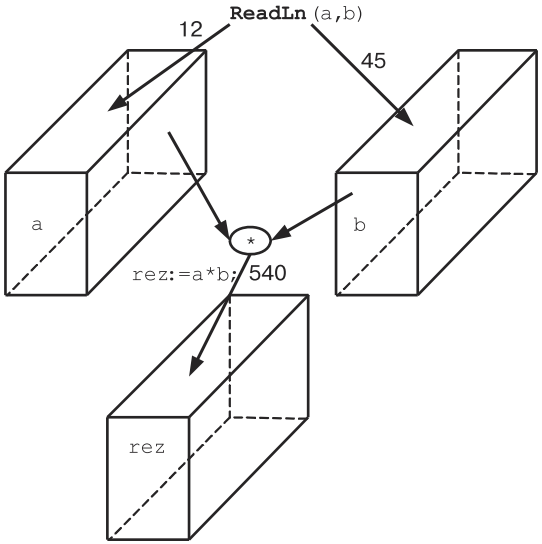


Рис. 1.2. Схема выполнения операторов **ReadLn**(a,b) и rez:=a*b

ременная *rez* получит значение, равное произведению числа *a* на число *b* (см. рис. 1.2). Так как в результате умножения двух целых чисел получается целое число, то переменная *rez* описана типом Integer (значениями которого могут быть лишь целые числа).

Следующий оператор — это снова оператор **WriteLn**`('Их произведение равно',rez);` — он выведет на экран текст, заключенный между апострофами, а за ним значение переменной *rez*.

Затем следующий оператор WriteLn выведет на экран сообщение: `Нажмите Enter`, а оператор ReadLn будет ожидать этого нажатия в окне выполнения.

В конце раздела операторов стоит служебное слово End, после которого стоит точка.

Запуск программы. Для того чтобы запустить программу, выходим в главное меню (нажатием F10) — первое окно, выбираем режим Run и дважды нажимаем Enter. На экране появляется сообщение:

```
Введите два числа через пробел
```

Курсор мигает в следующей строке, вводим в нее два целых числа через пробел и нажимаем Enter, после этого появляется сообщение:

```
Их произведение равно ...
Нажмите Enter
```

Вместо точек будет написано значение переменной *rez*, то есть число, равное произведению первого введенного числа на второе. Это сообщение будет на экране до тех пор, пока не нажата клавиша Enter.

Сохранение программы. Для того чтобы сохранить программу на внешнем носителе, необходимо:

- выйти в главное меню и выбрать режим **File**;
- нажать Enter, из появившегося окна выбрать режим **Save as...** и нажать клавишу Enter;
- в появившемся окне набрать имя файла и нажать клавишу Enter. Например, f:\prim1_1.pas; здесь f:\ – это логическое имя диска, на котором будем сохранять файл, prim1_1 — имя файла (оно может содержать не более 8 символов), pas — расширение, сообщающее о том, что файл содержит программу, написанную на языке Паскаль.

Примечания

1. Список символов, которые нельзя употреблять в именах файлов: *, =, +, [,], \, |, :, ., <, >, /, ?. Также запрещены запятая, пробел и буквы русского алфавита.
2. Для быстрого сохранения файла можно воспользоваться командами **Save** или **Save all** меню **File**.

Выход из системы программирования. Для того чтобы закончить работу, необходимо:

- выйти в главное меню и выбрать режим **File**;
- нажать Enter и из появившегося окна выбрать режим **Quit**, после чего нажать либо Enter, либо комбинацию клавиш Alt–X.

Экспериментальный раздел занятия

1. Введите числа при работе программы, например 4567 и 789. Убедитесь, что у вас получается неправдоподобный результат — отрицательное число (–1117). Найдите экспериментальным путем такой интервал значений переменных *a* и *b*, при котором результат умножения правильный.

2. Введите вместо числа какой-нибудь символ. Убедитесь, что выполнения программы не произошло, компьютер выдал сообщение об ошибке `Error 106: Invalid numeric format`.

3. Добавьте лишний знак апострофа в операторе WriteLn. Убедитесь, что компиляция программы не произошла, а компьютер вас порадовал ошибкой `Error 8: String constant exceeds line`.

4. Измените в программе My1_1 оператор `rez:=a*b` на `rez:=a-(a Div b)*b`. Выясните действие операции Div для переменных целого типа. Измените текстовую часть следующего за оператором присвоения оператора WriteLn, отразив в ней результат вашего исследования.

5. Добавьте в программу предыдущего примера переменную с именем *ost*, оператор присвоения `ost:=a Mod b` и оператор вывода `WriteLn('???????? ',ost);`. Выясните, какое действие выполняется с помощью оператора Mod. Замените знаки вопроса вашими пояснениями его работы.

6. Наберите следующую программу.

```
Program My1_2;
 Var a: Integer;
 Begin
  WriteLn('Введите целое число');
```

```
  ReadLn(a);
  WriteLn('???????? ', Abs(a));
  WriteLn('Нажмите Enter');
  ReadLn;
End.
```

Вашей задачей является замена знаков вопроса в операторе WriteLn на текст, поясняющий работу операции Abs. Выполните аналогичное исследование для операций Sqr(a), Ord(a), Succ(a), Pred(a).

Примечание

Рекомендуется следующий порядок работы. Текст программы My1_1 сохраняется под новым именем, например Prim1_2.pas, изменяется в соответствии с заданием и затем вновь сохраняется. Это позволит сократить время на набор программы.

Задания для самостоятельной работы

1. Измените программу для нахождения суммы двух чисел.
2. Измените программу для нахождения суммы четырех чисел.
3. Найдите значение выражения $(a+(d-12) \cdot 3) \cdot (c-5 \cdot k)$, где значения переменных a, d, c и k вводятся с клавиатуры.
4. Взяв за образец рис. 1.2, нарисуйте схему, иллюстрирующую выполнение следующего фрагмента программы:

```
x:=13;
y:=25;
t:=x;
x:=y;
y:=t;
```

5. Измените рисунок из предыдущего примера, заменив три последних оператора на:

```
x:=x-y;
y:=x+y;
x:=y-x;
```

6. Напишите программу вывода на экран нескольких чисел в виде

```
13
   14
      15
         16
```

или

```
101
    102
        103
```

Справочные сведения

1. В системе программирования есть встроенный редактор текста — режим работы **Edit** (главное меню). Мы не будем приводить команды управления курсором, вставки, удаления элементов текста и т. д., считая их традиционными и совпадающими с аналогичными командами текстовых процессоров.

2. Данные — общее понятие для всего того, с чем работает компьютер. В аппаратуре все данные представляются как последовательности двоичных цифр (разрядов).

В языках высокого уровня, а к ним относится Паскаль, **абстрагируются** от деталей представления данных в памяти компьютера. Любой тип данных определяет множество значений, которые может принимать величина этого типа, и те операции, которые можно применять к величинам этого типа. В Паскале работают с пятью типами данных: простыми, строковыми, составными, ссылочными и процедурными. К простым типам данных относятся целый, вещественный, логический, символьный, перечислимый и ограниченный (два последних определяются пользователем). На простых типах данных, кроме вещественного, определено *отношение порядка*. Что это такое? Все множество значений типа рассматривается как упорядоченное множество, и каждое значение связано с некоторым целым числом, которое есть его порядковый номер. В любом порядковом типе для каждого значения, кроме первого, существует предшествующее значение, и для каждого значения, за исключением последнего, существует последующее значение. Определены следующие стандартные функции для работы с порядковыми типами:

- Ord — возвращает порядковый номер для заданного значения любого порядкового типа.
- Pred — возвращает предшествующее значение для заданного значения порядкового типа (а если заданное значение первое?).

- Succ — возвращает следующее значение для заданного значения порядкового типа (а если заданное значение последнее?).

В Паскале есть возможность **конструировать** (создавать) свои типы данных из имеющихся типов, причем весьма сложные.

Итак, абстрагирование и конструирование суть *концепции типа данных*.

3. Каждая программа взаимодействует с окружающей средой с помощью операторов ввода и вывода. Если трактовать термин «программа» достаточно вольно, то можно считать, что любая программа что-то откуда-то берет, что-то делает с введенными данными (преобразует) и затем выводит куда-то полученные результаты. В Паскале связь программы с внешними устройствами осуществляется через имена файлов. Самый простой случай, когда эти имена связаны с клавиатурой (для ввода данных) и с экраном дисплея (для вывода).

Для ввода с клавиатуры используется оператор Read или ReadLn.

Вызов: Read(r_1,r_2,...,r_n).
Параметры: r_1,r_2,...,r_n имеют тип Integer, или Real, или Char, или String.
Действие, если r_i имеет тип:

- Integer или Real, то считывается одно число и значение его присваивается переменной r_i. При этом знаки пробела и перевода строки перед числом игнорируются;
- Char, то считывается один символ и присваивается переменной r_i;
- String, то считывается максимум q символов при длине q строковой переменной r_i.

Оператор ReadLn действует так же, как и Read. Отличие в том, что после ввода осуществляется переход на начало следующей строки.

Вывод на экран осуществляется с помощью операторов Write или WriteLn.

Вызов: Write(r_1,r_2,...,r_n).
Параметры: r_1,r_2,...,r_n имеют тип Integer, или Real, или Boolean, или Char, или String.
Действие: на экран выводится значение r_i в стандартном формате.
Работа WriteLn отличается тем, что после вывода осуществляется переход на начало следующей строки.

Занятие № 2. Целый тип данных

План занятия

1. Целый тип данных.
2. Разбор программы выделения цифр из десятичного числа.
3. Эксперименты с программами (перевод числа из десятичной системы счисления в двоичную систему счисления; выяснение сути арифметических операций с переменными целого типа; преобразование переменных целого типа).
4. Выполнение самостоятельной работы.

Целый тип данных

Существует пять целых типов: ShortInt, Integer, LongInt, Byte, Word (табл. 1.1). Они отличаются диапазоном значений, а значит, и размером памяти, отводимой для их представления.

Таблица 1.1

Тип	Диапазон значений	Объем памяти
ShortInt	–128 ... 127	1 байт, со знаком
Integer	–32768 ... 32767	2 байта, со знаком
LongInt	–2147483648 ... 2147483647	4 байта, со знаком
Byte	0 ... 255	1 байт, без знака
Word	0 ... 65535	2 байта, без знака

Операции с величинами целого типа: сложение (+), вычитание (–), умножение (*), нахождение целой части от деления (Div), нахождение остатка от деления (Mod). Так как целый тип данных относится к типам, на которых определено отношение порядка, то работают стандартные функции Ord, Succ и Pred.

Примечание

Переменной целого типа нельзя присваивать значение результата обычной операции деления «/». Убедитесь в этом с помощью простой модификации программы первого занятия. Попробуйте найти объяснение этому факту.

Возникают по крайней мере два достаточно сложных (на этой стадии освоения языка) вопроса:

- Почему при представлении целых чисел со знаком диапазон отрицательных чисел на одно значение больше диапазона положительных чисел?
- Как выполняются операции, например, при вычислении выражений, если все величины имеют разные целые типы?

Разбор программы выделения цифр из десятичного числа

Перед разбором программы следует рассмотреть выполнение операций Div и Mod (примеры приведены в табл. 1.2).

Таблица 1.2

19 Div 4=4	19 Mod 4=3
19 Div –4=–4	19 Mod –4=3
–19 Div 4=–4	–19 Mod 4=–3
–19 Div –4=4	–19 Mod –4=–3

В программе определяются цифры трехзначного числа. Можно ее использовать и для определения цифр двузначного числа, просто цифра сотен в этом случае равна нулю.

```
Program My2_1;
 Var a, one, dec, hun, rez:Integer;
 Begin
  WriteLn('Введите число');
  ReadLn(a);
  one:=a Mod 10;
  WriteLn('Цифра единиц числа - ',one);
  dec:=(a Div 10) Mod 10;
  WriteLn('Цифра десятков числа - ',dec);
  hun:=a Div 100;
  WriteLn('Цифра сотен числа - ',hun);
  rez:=hun*100+dec*10+one;
```

```
 WriteLn('А это тоже число - ',rez);
 Write('Enter ');
 ReadLn;
End.
```

Например, если вы введете число 137, то значение переменной *one* будет равно 7, *dec* – 3 и *hun* – 1. Вспомните деление чисел столбиком.

Примечание

Не забудьте сохранить программу под именем prim2_1.pas.

Экспериментальный раздел занятия

1. Измените программу My2_1 для нахождения цифр двузначного числа. Сохраните ее под именем Prim2_2.pas.

2. Измените программу My2_1 для нахождения цифр четырехзначного числа. Сохраните ее под именем Prim2_3.pas.

3. Деление на 10 и нахождение остатков от деления нам известны. Рассмотрите пример деления столбиком на 2. Остатки от деления в данном случае — или 0, или 1.

```
137 | 2
  1  68 | 2
     0  34 | 2
        0  17 | 2
           1  8 | 2
              0  4 | 2
                 0  2 | 2
                    0  1
```

Наберите следующую программу, отладьте ее (найдите и исправьте ошибки) и попробуйте дать объяснение полученному результату. Измените программу так, чтобы она правильно работала, например, с числом 115.

```
Program My2_2;
 Var rez:Integer;
 Begin
  WriteLn('137');
  WriteLn('10001001');
  Rez:=1*128+0*64+0*32+0*16+1*8+0*4+0*2+1*1;
  WriteLn(rez);
  ReadLn;
 End.
```

4. Наберите следующую программу:

```
Program My2_3;
 Uses Crt;
 Var a:Integer; b:Word; r1:Integer; r2:LongInt;
 Begin
```

```
  ClrScr; {Очистка экрана, процедура модуля Crt}
  a:=32000;b:=64000;
  r1:=a+b;
  WriteLn(r1);
  r2:=a+b;
  WriteLn(r2);
  ReadLn;
End.
```

После запуска вы увидите, что значение переменной *r1* равно 30464, а значение переменной *r2* – 96000. Если изменить тип переменной *r1* на Word, то результат не изменится. Используя информацию из табл. 1.1, измените программу так, чтобы проделать аналогичные эксперименты с другими целыми типами. Попробуйте найти логику получения результата компьютером.

Примечание

В фигурных скобках записываются комментарии; они не отображаются на экране и не исполняются программой.

5. Добавьте в программу My2_3 перед оператором ReadLn следующие два оператора:

```
WriteLn(LongInt(100*a));
WriteLn(100*LongInt(a));
```

Функция LongInt преобразует переменную типа Integer в тип LongInt. В первом случае преобразование выполняется после умножения, во втором — перед умножением. В первом случае получен результат, далекий от истины: отрицательное число –11264, во втором правильный: 3200000. Приведем основные правила, по которым в Паскале выполняются операции с переменными целых типов.

Перед выполнением операций (бинарных) над двумя операндами оба операнда преобразуются к общему для них типу. Им является тип с наименьшим диапазоном, включающим все возможные значения обоих типов. Например, общим типом для Integer и Byte будет Integer, для Integer и Word — LongInt. Результат будет общего типа.

Выражение в правой части оператора присвоения вычисляется независимо от размера или типа переменной в левой части!

Перед выполнением любой арифметической операции любой операнд длиной в 1 байт преобразуется в промежуточный операнд длиной в 2 байта, который является совместимым как с Integer, так и с Word.

Вопросы и задания для самостоятельной работы

1. Чему равны значения переменных *a* и *b* после выполнения последовательности действий?
   - ```
     a:=15 Div (16 Mod 7);
     b:=34 Mod a * 5 - 29 Mod 5 * 2;
     ```
   - ```
     a:=4 * 5 Div 3 Mod 2;
     b:=4 * 5 Div ( 3 Mod 2);
     ```
2. Дано двузначное число. Определите:
   - сумму и произведение цифр числа;
   - число, образованное перестановкой цифр исходного числа.
3. Дано трехзначное число. Определите:
   - сумму и произведение цифр числа;
   - число, образованное перестановкой цифр исходного числа;
   - число, полученное перестановкой цифр десятков и единиц;
   - число, полученное перестановкой цифр сотен и десятков;
   - четырехзначное число, полученное приписыванием цифры единиц в качестве цифры тысяч (например, из числа 137 необходимо получить число 7137).

   Сколько различных чисел можно получить из трехзначного числа путем перестановки цифр?
4. Решите задачу 3 (кроме последнего пункта) для четырехзначных чисел.

   Предложите максимальное количество разумных модификаций рассматриваемой задачи.
5. Арифметическая прогрессия — это последовательность чисел, в которой разность между последующим и предыдущим элементами остается неизменной. Последовательность 12, 15, 18, 21, 24, ... является арифметической прогрессией, 12 — первый член прогрессии (a_1), разность

прогрессии равна 3. Любой член прогрессии вычисляется по формуле $a_n=a_1+d\cdot(n-1)$, где d — разность прогрессии, n — номер взятого члена. Даны a_1 и d. Найдите n, при котором значение a_n выходит за диапазон типа Integer (экспериментальным путем).

6. Сумма первых n членов арифметической прогрессии вычисляется по формуле $S_n=(a_1+a_n)\cdot n/2$. Даны a_1 и d. Найдите n, при котором значение S_n выходит за диапазон типа Integer (экспериментальным путем).

Справочные сведения

Выражения состоят из операций и операндов. Различают бинарные операции — выполняемые над двумя операндами, и унарные (одноместные) — выполняемые над одним операндом. Бинарные операции записываются в обычной математической форме, знак унарной операции предшествует операнду. Порядок выполнения операций в выражении определяется **правилами приоритета** (табл 1.3).

Таблица 1.3

Операции	Приоритет	Тип операции
@, Not, +, –	1-й (высший)	унарный
*, /, Div, Mod, And, Shl, Shr	2-й	мультипликативный
+, –, Or, Xor	3-й	аддитивный
=, <>, <, >, <=, >=, In	4-й (низший)	операции отношения

Примечание

Если некоторые из операций вам не очень понятны, не отчаивайтесь. Всему свое время.

Основные правила приоритета:

- операнд, расположенный между знаками двух операций с разными приоритетами, служит границей операции с более высоким приоритетом;
- операнд, расположенный между двумя операциями с равными приоритетами, является границей для операции, расположенной слева;
- выражения в скобках вычисляются прежде, чем они будут обрабатываться как один операнд;
- операции с равными приоритетами обычно выполняются слева направо.

Занятие № 3. Команды редактора для работы с блоками, работа с окнами[5]

План занятия

1. Команды редактора для работы с блоками.
2. Работа с окнами.
3. Эксперименты с программами (вычисление степеней двойки; перевод чисел из десятичной системы счисления в двоичную систему счисления).
4. Выполнение самостоятельной работы.

Команды редактора для работы с блоками

Блок — любой фрагмент текста программы. Одновременно в тексте может существовать только один блок. Отмеченный блок выделяется на экране яркостью. Для того чтобы что-то сделать с блоком, его необходимо выделить, то есть отметить начало и конец блока. После этого выполняются действия с блоком.

В табл. 1.4 приведены основные команды редактора (режим **Edit** главного меню) для работы с блоками.

Таблица 1.4

Ctrl	+	K	+	B	Отметить начало блока
Ctrl	+	K	+	K	Отметить конец блока
Ctrl	+	K	+	T	Отметить одно слово как блок
Ctrl	+	K	+	C	Скопировать блок
Ctrl	+	K	+	Y	Удалить блок
Ctrl	+	K	+	H	Убрать выделение блока
Ctrl	+	K	+	V	Переместить блок
Ctrl	+	K	+	R	Ввести с диска ранее записанный блок, начиная с позиции курсора
Ctrl	+	K	+	W	Записать отмеченный блок на диск

Примечания

1. Последовательность выполнения команд:
   - установить курсор в требуемое место текста;
   - нажать клавишу Ctrl;
   - не отпуская клавишу Ctrl, нажать клавишу с буквой K;
   - отпустив клавишу с буквой K, не отпуская клавишу Ctrl, нажать клавишу со второй буквой.

[5] Если команды работы с блоками и окнами известны, то можно ограничиться экспериментальной частью занятия.

2. Знание команд работы с блоками значительно ускоряет набор ваших программ. Используйте их при работе.

Упражнения для самостоятельной работы

1. Наберите любой содержательный текст. Например:

«На счетах десятичное число кодируется положением костяшек на спицах, а в арифмометре (механическом вычислителе) число кодируется положением диска с 10 зубцами. С помощью таких связанных между собой дисков можно построить суммирующее устройство.

Идея такого устройства принадлежит Леонардо да Винчи (1452–1519). А впервые такое устройство сделал в 1642 году французский философ и математик Блез Паскаль (1623–1662).

Выдающийся американский математик Джон фон Нейман (1905–1957) при работе в составе группы исследователей над машиной «ЭНИАК» сформулировал принципы, лежащие в основе функционирования современных вычислительных машин. Суть этих принципов: в памяти ЭВМ хранятся не только данные, но и обрабатывающая их программа; представление в памяти данных и программы совпадает; программа должна выполняться последовательно команда за командой».

2. Выделите его фрагменты как блоки.
3. Скопируйте фрагменты текста (выделяется блок, курсор устанавливается в требуемое место, выполняется команда копирования).
4. Переместите фрагменты текста (выделяется блок, курсор устанавливается в требуемое место, выполняется команда перемещения).
5. Запишите фрагменты текста через диск (выделяется блок, задается команда записи на диск, при этом необходимо определить имя блока, курсор устанавливается в требуемое место, выполняется команда чтения блока с диска).

Работа с окнами

В Паскале есть возможность работы с несколькими окнами (режим **Window** главного меню). Окно — прямоугольная область экрана. Его размеры и положение на экране можно изменять. Программа, а она хранится на диске как файл с определенным именем, размещается в определенном окне. Активным в текущий момент является только одно окно, в нем находится курсор.

Рассмотрим команды для работы с окнами (табл. 1.5).

Таблица 1.5

Команда	Функциональная клавиша			Назначение
Tile				Последовательное размещение окон
Cascade				Каскадное размещение окон
Close All				Закрытие всех окон
Size/Move	Ctrl	+	F5	Изменение размера, перемещение окна. При выполнении команды изменяется цвет рамки окна. Для изменения размера окна используется комбинация клавиш Shift+⇑, ⇓, ⇐, ⇒. Для перемещения – ⇑, ⇓, ⇐, ⇒. Для завершения работы с окном следует нажать клавишу Enter. Цвет рамки окна изменится на первоначальный
Zoom			F5	Размеры активного окна устанавливаются равными полному экрану
Next			F6	Переход к следующему окну
Previos	Shift	+	F6	Переход к предыдущему окну
Close	Alt	+	F3	Закрытие активного окна
List				Просмотр списка открытых окон

Упражнения для самостоятельной работы

1. Откройте все программы предыдущих занятий.
2. Выполните перечисленные выше команды работы с окнами.

Экспериментальный раздел занятия

1. Выполните обычные действия с программой My3_1 (набор, компиляцию, запись на диск, запуск).

```
Program My3_1;
 Uses Crt;
 Var r0,r1,r2,r3,r4,r5,r6:Integer;
 Begin
  ClrScr;
  WriteLn('Вычисляем степени числа 2');
  r0:=1;
```

```
  r1:=2;
  r2:=r1*r1;
  r3:=r2*r1;
  r4:=r2*r2;
  r5:=r1*r1*r1*r1*r1;
  r6:=r3*r3;
  WriteLn('Нулевая степень ',r0);
  WriteLn('Первая степень ',r1);
  WriteLn('Вторая степень ',r2);
  WriteLn('Третья степень ',r3);
  WriteLn('Четвертая степень ',r4);
  WriteLn('Пятая степень ',r5);
  WriteLn('Шестая степень ',r6);
  ReadLn;
 End.
```

Примечание

При наборе программы рекомендуется использовать изученные команды работы с блоками. Набирается `r2:=r1*r1`, а затем этот фрагмент выделяется как блок, копируется 4 раза и модифицируется. Аналогичные действия выполняются с операторами WriteLn.

Давайте подсчитаем, какое количество операций требуется, например, для вычисления 5-й степени двойки в нашей программе — 4 операции. А можно ли обойтись меньшим количеством операций? Оказывается, можно: `r5:=r2*r2*r1`. Всего 3 операции — одна операция умножения для вычисления *r*2 и еще две для вычисления результата.

Модифицируйте программу так, чтобы она вычисляла степени двойки до 10-й включительно и притом так, чтобы каждая степень двойки вычислялась за минимальное количество операций.

2. Выполните обычные действия с программой My3_2 (набор, компиляцию, запись на диск, запуск).

Примечание

При наборе программы рекомендуется использовать изученные команды работы с блоками и окнами. Скопируйте через диск текст программы My3_1, запишите его как текст программы My3_2, а затем измените его, максимально используя команды работы с блоками. Например, набирается `s0:=b Mod 2; b:=b Div 2;` и копируется столько раз, сколько необходимо.

```
Program My3_2;
 Uses Crt;
  Var r0,r1,r2,r3,r4,r5,r6:Integer;
    s0,s1,s2,s3,s4,s5,s6:Integer;
    a,b,rez:Integer;
    one,dec,hun:Integer;
 Begin
  ClrScr;
  {Вычисление степеней двойки}
  r0:=1;r1:=2;
  r2:=r1*r1;
  r3:=r2*r1;
  r4:=r2*r2;
  r5:=r3*r2;
  r6:=r3*r3;
  WriteLn('Введите число меньше 128 и больше 64');
  ReadLn(a);
  b:=a;
  {Перевод числа в двоичную систему счисления}
  s0:=b Mod 2; b:=b Div 2;
  s1:=b Mod 2; b:=b Div 2;
  s2:=b Mod 2; b:=b Div 2;
  s3:=b Mod 2; b:=b Div 2;
  s4:=b Mod 2; b:=b Div 2;
  s5:=b Mod 2; b:=b Div 2;
  s6:=b Mod 2; b:=b Div 2;
  b:=a;
  {Выделение десятичных цифр в записи числа}
  one:=b Mod 10; b:=b Div 10;
  dec:=b Mod 10; b:=b Div 10;
  hun:=b Mod 10;
  WriteLn('Мы правильно выделили десятичные цифры');
  WriteLn('Вывод числа в десятичной системе
          счисления',hun*100+dec*10+one);
  WriteLn(hun,dec,one);
  WriteLn('Это же число в двоичной системе
          счисления');
  WriteLn(s6,s5,s4,s3,s2,s1,s0);
  WriteLn('Переводим число из двоичной системы
          счисления в десятичную');
```

```
  {Перевод}
  rez:=s6*r6+s5*r5+s4*r4+s3*r3+s2*r2+s1*r1+s0*r0;
  WriteLn('Все сделано правильно, числа совпадают');
  WriteLn(a,' ',rez);
  ReadLn;
End.
```

Введите числа больше 128, но меньше 256. Оцените результат работы программы. Модифицируйте программу так, чтобы и в этом случае она была работоспособна.

Задания для самостоятельной работы

1. Модифицируйте программу My3_1 так, чтобы она вычисляла степени тройки.
2. Модифицируйте программу My3_2 так, чтобы она выполняла перевод десятичных чисел из определенного интервала в троичную систему счисления и обратно.
3. Подсчитайте сумму пятеричных чисел в интервале от 20_5 до 40_5, включая границы интервала.
4. Восстановите цифры двоичного числа, на месте которых записан символ «*»: $1**1_2+0011_2=1100_2$. Придумайте и выполните еще несколько примеров такого типа.
5. Составьте таблицы сложения и умножения в четверичной и восьмеричной системах счисления.

Занятие № 4. Логический тип данных, операции сдвига

План занятия

1. Логический тип данных.
2. Операции сдвига.
3. Эксперименты с программами (построение таблиц истинности; выполнение операций сдвига и логических операций).
4. Выполнение самостоятельной работы.

Логический тип данных

Переменные логического типа описываются с помощью идентификатора Boolean. Диапазон их значений — два: False (ложь) или True (истина), размер выделяемой памяти — 1 байт

(False и True — стандартные константы). Тип является перечислимым, поэтому: False<True, Ord(False)=0, Ord(True)=1, Succ(False)=True, Pred(True)=False.

Перечислим четыре логические операции, реализованные в Паскале: отрицание — Not; логическое умножение, или конъюнкция, — And; логическое сложение, или дизъюнкция, — Or («или»); исключающее «или» (сложение по модулю два) — Xor. Результаты выполнения этих операций над переменными логического типа *x* и *y* приведены в табл. 1.6.

Таблица 1.6

Значение операнда		**Значение операции**			
x	*y*	Not *x*	*x* And *y*	*x* Or *y*	*x* Xor *y*
False	False	True	False	False	False
False	True	True	False	True	True
True	False	False	False	True	True
True	True	False	True	True	False

Выше приведены четыре таблицы истинности (сведены в одну таблицу), с помощью которых в математической логике обычно описываются значения логических функций.

Таблица истинности представляет собой таблицу, устанавливающую соответствие между возможными значениями наборов переменных и значениями функции.

Следует четко понимать, что результатом выполнения операций сравнения (отношения) «<» (меньше), «>» (больше), «<=» (меньше или равно), «>=» (больше или равно), «<>» (не равно), «=» (равно) является величина логического типа. Ее значение равно True, если отношение выполняется для значений входящих в него операндов, и False — в противном случае.

В языке Паскаль нет возможности ввода логических данных с помощью оператора Read. Однако предусмотрен вывод значений переменных логического типа с помощью оператора Write.

Операции сдвига

Речь идет о двух операциях: ShL — сдвиг влево и ShR — сдвиг вправо. Тип операндов и результата в операциях сдвига — Integer. Итак, *m* ShL *n* — значение *m* сдвигается влево на *n* разрядов; *m* ShR *n* — значение *m* сдвигается вправо на *n* разрядов. При выполнении операций разряды, вышедшие

за пределы области памяти, выделяемой для указанного типа данных, теряются, а с другой стороны добавляются нули. Например, если m равно 32, то сдвиг влево на один разряд дает 64, а сдвиг вправо — 16. Операции равносильны умножению и делению на два.

Экспериментальный раздел занятия

1. Выполните обычные действия с программой My4_1 (набор, компиляцию, запись на диск, запуск).

```
Program My4_1;
 Uses Crt;
 Var a,b:Boolean;
 Begin
  ClrScr;
  a:=True;b:=True;WriteLn(a:6,b:6,a And b:6);
  a:=True;b:=False;WriteLn(a:6,b:6,a And b:6);
  a:=False;b:=True;WriteLn(a:6,b:6,a And b:6);
  a:=False;b:=False;WriteLn(a:6,b:6,a And b:6);
  ReadLn;
 End.
```

Примечание

При наборе программы не забывайте использовать команды работы с блоками. Набирается `a:=True; b:=True; WriteLn(a:6,b:6,a And b:6);`, а затем этот фрагмент выделяется как блок, копируется 3 раза и модифицируется.

Измените программу для проверки остальных рассмотренных выше логических операций.

2. Выполните обычные действия с программой My4_2 (набор, компиляцию, запись на диск, запуск).

```
Program My4_2;
 Uses Crt;
 Var m,n:Integer;
 Begin
  ClrScr;
  WriteLn('Введите число и количество сдвигов');
  ReadLn(m,n);
  WriteLn(' При сдвиге на ',n,' разрядов влево
            числа ',m,' получаем число ',m ShL n);
  WriteLn('Введите число и количество сдвигов');
```

```
 ReadLn(m,n);
 WriteLn(' При сдвиге на ',n,' разрядов вправо
          числа ',m,' получаем число ',m ShR n);
 ReadLn;
End.
```

Введите в первом и во втором случаях числа 31 и 1; убедитесь, что получаются, соответственно, числа 64 и 16. Сдвиги вправо отрицательных чисел приводят к интересным результатам. Например, если вы введете –1 и 1 для первого и второго сдвигов, то получите –2 и 32767. Если первый результат вполне объясним, то второй требует вспомнить о представлении отрицательных целых чисел в дополнительном коде. Пусть у нас не шестнадцать разрядов для представления чисел (тип Integer), а 4. Представление –1 в дополнительном коде есть 1111_2. Сдвиг вправо на один разряд приводит к числу 0111_2, а это не что иное, как 7_{10}.

3. Выполните набор программы My4_3, набрав сначала только первые три оператора WriteLn. Откомпилируйте ее, запишите на диск.

```
Program My4_3;
 Uses Crt;
 Begin
  ClrScr;
  WriteLn(1365 And 2730);
  WriteLn(1365 Or 2730);
  WriteLn(1365 Xor 2730);
  WriteLn(1365 And $FF);
  WriteLn(1365 And $FF0);
  WriteLn(1365 And $FF00);
  ReadLn;
 End.
```

Мы видим, что с величинами типа Integer можно выполнять логические операции: они выполняются поразрядно над двоичными представлениями чисел. Почему для данного эксперимента выбраны числа 1365 и 2730? Двоичное представление этих чисел имеет вид: $1365_{10}=010101010101_2$, $2730_{10}=101010101010_2$ (рассматриваются только 12 младших разрядов). Операция And дает в результате число 0, а операции Or и Xor — 4095_{10}. Поэкспериментируйте с этой версией программы. Убедитесь, например, что –256 And 256=0, а –256 Or

256=–1 и –256 Xor 256=–1. Попытайтесь дать разумное объяснение этому результату.

Добавьте к программе следующие три оператора WriteLn. В шестнадцатеричной системе счисления для обозначения чисел 10, 11, 12, 13, 14, 15 одним символом используются соответственно буквы латинского алфавита A, B, C, D, E, F. Таким образом, двоичное представление, например, F — 1111_2. Знак $ означает, что величина (константа) записана в шестнадцатеричной системе счисления. Запустите программу. Убедитесь в том, что результаты равны 85, 1360, 1280. Его правильность подтверждается выделением соответствующих разрядов из числа 010101010101_2 и переводом остатка в десятичную систему счисления. Исследуйте описанным способом представление в дополнительном коде отрицательных целых чисел.

Задания для самостоятельной работы

1. В математической логике известна функция следования, или импликация, ($x \Rightarrow y$), ее таблица истинности представлена в табл. 1.7.

Таблица 1.7

x	*y*	$x \Rightarrow y$
False	False	True
False	True	True
True	False	False
True	True	True

Проверьте, что $x \Rightarrow y$ эквивалентно Not(x) Or y. Составьте программу проверки эквивалентности этих двух логических функций.

2. В математической логике известна функция Шеффера ($x \mid y$), ее таблица истинности представлена в табл. 1.8.

Таблица 1.8

x	*y*	$x \mid y$
False	False	True
False	True	True
True	False	True
True	True	False

Проверьте, что $x \mid y$ эквивалентно Not(x) Or Not(y). Составьте программу проверки эквивалентности этих двух логических функций.

3. В математической логике известна функция Вебба, или стрелка Пирса, ($x \Downarrow y$), ее таблица истинности приведена в табл. 1.9.

Таблица 1.9

x	*y*	*x⇓y*
False	False	True
False	True	False
True	False	False
True	True	False

Проверьте, что $x \Downarrow y$ эквивалентно Not(x) And Not(y). Составьте программу проверки эквивалентности этих двух логических функций.

4. Постройте таблицы истинности для следующих функций:
   - $(x \mid y) \mid z$;
   - $(x \Downarrow y) \Downarrow z$;
   - ($x \Rightarrow y$) And z;
   - Not (x Or Not(y) And z);
   - x And Not(y Or Not(z));
   - Not(Not(x) Or y And z).

Занятие № 5. Составной оператор и оператор If – Then – Else

План занятия

1. Составной оператор.
2. Условный оператор.
3. Эксперименты с программами (нахождение наибольшего из двух и трех чисел).
4. Выполнение самостоятельной работы.

Составной оператор

Составной оператор представляет собой последовательность операторов, выполняемых в том порядке, в котором они записаны в программе. Его схема:

```
Begin
 оператор;
 оператор;
 …
 оператор;
End;
```

Примечание

Разделитель «;» перед End можно не записывать.

Условный оператор

Условный оператор может записываться в полной и неполной формах.

Полная форма условного оператора:

```
If <логическое выражение> Then <оператор 1>
 Else <оператор 2>;
```

Неполная форма условного оператора:

```
If <логическое выражение> Then <оператор>;
```

Выполнение условного оператора начинается с вычисления значения логического выражения, записанного в условии. Простые условия записываются в виде равенств или неравенств. Сложные условия составляются из простых с помощью логических операций. Как известно, значением логического выражения является или True, или False. В первом случае выполняется <оператор 1>, во втором — <оператор 2>. В качестве <оператор 1> или <оператор 2> может выступать любой оператор языка программирования Паскаль, в частности и составной оператор, и условный оператор. В последнем случае получаемая конструкция называется вложенными условными операторами. Допускается запись неполного условного оператора, без ветви Else. В этом случае при значении False никаких действий не производится. В записи условных операторов, в случае их вложенности, возникает неоднозначность типа:

```
If <условие 1> Then <оператор 1>
If <условие 2> Then <оператор 2> Else <оператор 3>;
```

Неясно, к какому оператору If относится ветвь Else. Такая неоднозначность разрешается по следующему правилу: Else относится к ближайшему оператору If, у которого еще отсутствует данная ветвь.

Экспериментальный раздел занятия

1. Разбор условного оператора можно выполнить на следующем простом примере: вывести на экран наибольшее из двух данных чисел.

```
Program My5_1;
 Var x,y: Integer;
 Begin
  WriteLn('Введите 2 числа');
  ReadLn(x,y);
  If x>y Then WriteLn(x)Else WriteLn(y);
  ReadLn;
 End.
```

Поставьте «;» после оператора WriteLn(x). Убедитесь, что появилась ошибка `Error 113: Error in statement`. Конструкция (оператор) If - Then - Else неделима, поэтому разделитель «;» недопустим. В случае равенства чисел ваша программа выводит значение переменной y. Измените программу так, чтобы в этом случае она выводила на экран сообщение «Числа равны».

2. Попробуем найти наибольшее из трех данных чисел — значения переменных x, y и z. Предположим, что все числа различны. Возможны шесть различных случаев, они приведены на рис. 1.3.

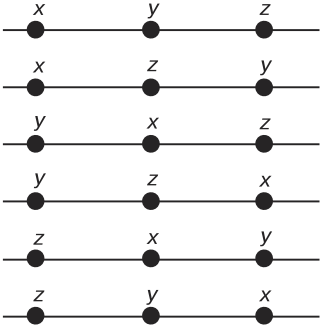


Рис. 1.3. Варианты взаимного расположения трех чисел

Программа определения наибольшего из трех чисел имеет вид:

```
Program My5_2;
 Var x,y,z:Integer;
 Begin
  WriteLn('Введите три числа через пробел');
  ReadLn(x,y,z);
  If (x>y) And (x>z) Then WriteLn(x)
   Else If (y>x) And (y>z) Then WriteLn(y)
        Else WriteLn(z);
   { If x>y Then If x>z Then WriteLn(x)
                 Else WriteLn(z)
     Else If y>z Then WriteLn(y)
          Else WriteLn(z); }
   ReadLn;
  End.
```

Вторая версия решения заключена в фигурные скобки (комментарии). Уберите их, включите первую реализацию как комментарий, убедитесь в правильности решения. Измените программу так, чтобы анализировался и случай равенства чисел. Обратите внимание на то, что при написании сложных условий простые условия заключаются в скобки. Это связано с тем, что операции сравнения имеют более низкий приоритет, чем логические операторы.

Вопросы и задания для самостоятельной работы

1. Имеется условный оператор:

```
If D<>10 Then WriteLn('ура! ')
 Else WriteLn('плохо...');
```

Можно ли заменить его следующими операторами?

```
If D=10 Then Writeln('ура!')
 Else Writeln('плохо...');
If Not(D=10) Then Writeln('ура!')
 Else Writeln('плохо...');
If Not(D=10) Then Writeln('плохо...')
 Else Writeln('ура!');
If Not(D<>10) Then Writeln('плохо...')
 Else Writeln('ура!').
```

2. Запишите условный оператор, с помощью которого значение переменной вычисляется по формуле $a+b$, если a — нечетное, и по формуле $a*b$, если a — четное.
3. Напишите условный оператор, с помощью которого вычисляется значение функции $y=\begin{cases} x^2+5 & \text{при } x>3; \\ x-8 & \text{при } x\le 3. \end{cases}$
4. Запишите условный оператор, который выводит на экран номер четверти, которой принадлежит точка с координатами (x,y), при условии что x и y отличны от 0.
5. Дано двузначное число. Напишите программу определения:
   - является ли сумма его цифр двузначным числом;
   - больше ли числа x сумма его цифр, число x вводится с клавиатуры;
   - кратна ли шести сумма его цифр;
   - больше ли цифра десятков цифры единиц;
   - оканчивается ли число цифрой 5.

   Придумайте не менее восьми аналогичных задач с трехзначными числами.
6. Напишите программу, подсчитывающую сумму только положительных из трех данных чисел.
7. Даны три числа. Напишите программу, подсчитывающую количество имеющихся среди них четных чисел.
8. Дано трехзначное число. Напишите программу определения — является ли оно палиндромом («перевертышем»), то есть числом, десятичная запись которого читается одинаково слева направо и справа налево.
9. Даны два целых числа m и n. Напишите программу, которая выводит на экран частное от деления m на n, если они делятся нацело, а в противном случае — сообщение «m на n нацело не делится».
10. Даны два целых числа n и m. Напишите программу, которая вычитает из первого числа 100, если оно больше второго по абсолютной величине.
11. Напишите программу вычисления значения функции:

$$y=\begin{cases} x-12 & \text{при } x>0; \\ 5 & \text{при } x=0; \\ x^2 & \text{при } x<0. \end{cases}$$

12. Даны три целых числа. Напишите программу нахождения среднего из них. Средним назовем число, которое больше наименьшего из данных чисел, но меньше наибольшего.

13. Напишите программу нахождения произведения двух наибольших из трех введенных с клавиатуры чисел.

14. Напишите программу нахождения количества положительных (отрицательных) чисел среди четырех целых чисел a, b, c и d.

15. Дано двузначное (трехзначное) число. Напишите программу определения:

- входит ли в него цифра 5;
- входят ли в него цифры 4 и 7;
- входят ли в него цифры 3, 5 и 7.

16. Даны целые числа x и y. Напишите программу определения знака разности $x-y$. Разность не вычислять. Разрешается сравнивать числа x и y с нулем, а между собой можно сравнивать только модули чисел x и y.

17. Известна текущая дата. Пользователь вводит день, месяц и год своего рождения. Напишите программу определения, исполнилось или нет пользователю полных 16 лет.

18. Напишите программу определения факта попадания точки $M(x,y)$ в заштрихованную область, изображенную на рис. 1.4.

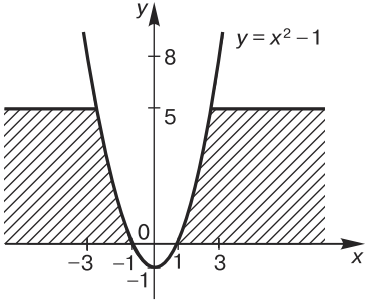


Рис. 1.4. Область возможной локализации заданной точки (к заданию № 18)

19. Напишите программу определения принадлежности точки $M(x,y)$ заштрихованной области, изображенной на рис. 1.5 (уравнение окружности имеет вид $x^2+y^2=r^2$).

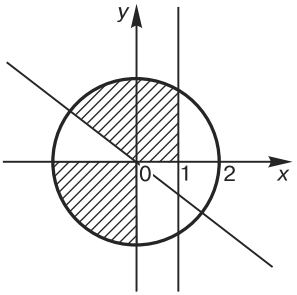


Рис. 1.5. Область возможной локализации заданной точки (к заданию № 19)

20. Напишите программы вычисления выражений

- $\max(x+y+z,\ xyz)+3$;
- $\min(x^2+y^2,\ y^2+z^2)-4$,

если x, y, z вводятся с клавиатуры.

21. Напишите программу, которая из трех введенных с клавиатуры чисел возводит в квадрат положительные, а отрицательные оставляет без изменения.

22. Даны два конверта прямоугольной формы с длинами сторон (a, b) и (c, d). Напишите программу, которая определяет, можно ли один из конвертов вложить в другой.

23. Напишите программу, которая определяет вид треугольника по длинам его сторон a, b и c (если соотношение заданных длин позволяет его построить). Как известно, условием «составимости» треугольника из отрезков заданной длины является одновременное выполнение следующих соотношений: $a+b>c$, $b+c>a$, $a+c>b$. Напомним также, что треугольники могут быть разными: равносторонними, равнобедренными, прямоугольными, равнобедренными прямоугольными и так далее. Наконец, следует учесть, что в качестве длин сторон могут быть случайно введены как нулевые, так и отрицательные значения.

Примечание

Эту хорошо известную задачу (из книги Г. Майерса «Искусство тестирования программ») иногда называют тестом Майерса на профессиональную пригодность. Если вы сможете реализовать в своей программе порядка 10 различных ситуаций, то вам можно уверенно выбрать программирование своей специальностью.

Занятие № 6. Оператор цикла For

План занятия

1. Оператор цикла For.
2. Эксперименты с программами (определение чисел-палиндромов; поиск чисел, содержащих ровно три одинаковые цифры).
3. Выполнение самостоятельной работы.

Оператор цикла For

Оператор For задает многократное выполнение некоторого другого оператора (который может быть и составным) с одновременным пошаговым изменением значения управляющей переменной.

Вид оператора For:

```
For <управляющая переменная>:=a To b Do <оператор>;
```

или

```
For <управляющая переменная>:=a DownTo b Do
<оператор>;
```

где a — начальное значение управляющей переменной, b — конечное значение управляющей переменной.

Начальное (a) и конечное (b) значения управляющей переменной могут быть представлены константами, переменными или арифметическими выражениями. Они определяются один раз в начале выполнения оператора For и не изменяются во время выполнения этого оператора. Управляющая переменная, а также a и b должны быть одного типа, обязательно порядкового.

Оператор после слова Do выполняется один раз для каждого значения управляющей переменной из диапазона, определяемого значениями a и b. Если в операторе For используется слово To, то значение управляющей переменной увеличивается на единицу при каждом повторении — a, $a+1$,..., $b-1$, b, а при использовании DownTo — уменьшается на единицу.

Если при использовании слова To окажется, что $a>b$, то оператор после слова Do («тело» цикла) не будет выполнен ни разу и выполнение цикла с параметром сразу же закончится (соответственно, при использовании DownTo это произойдет, если $a<b$).

Рекомендации по использованию

Оператор For применяют тогда, когда известно число повторений одного и того же действия (оператора). Изменение значения управляющей переменной в теле цикла может привести к ошибкам, считается «дурным тоном» в программировании, поэтому договоримся о том, что это действие запрещено законом, т. е. учителем. Управляющая переменная после выполнения оператора For имеет неопределенное значение (на самом деле она имеет значение *b*, но стандартом языка это не оговорено). Запретим использование ее значения для анализа чего-либо после выполнения оператора For, а также «искусственные» выходы из For с помощью операторов GoTo, Exit и т. п. Оператор For должен иметь одну точку входа и одну точку выхода.

Экспериментальный раздел занятия

1. Дано натуральное четырехзначное число n ($n \le 9999$). Определить, является ли оно палиндромом («перевертышем»), с учетом всех четырех цифр. Например, палиндромами являются числа: 2222, 6116, 7447.

Начнем не с программы, а с ручной трассировки логики решения. Это важно, очень важно. Мы с вами с помощью этого приема должны достичь того, чтобы при написании программы у вас одновременно складывался «зрительный образ» ее работы, чтобы вы видели ее работу, причем это должна быть не статическая «картинка», а динамическая. Трассировка обычно выполняется для конкретных значений входных параметров задачи.

Итак, у нас четырехзначное число, поэтому переменная оператора For (с именем *i*) изменяется от 1 до 4. В переменной с именем *m* хранится «остаток» числа; в первоначальный момент времени он равен введенному числу. В переменной с именем *r* формируем значение числа-«перевертыша». Основными

Таблица 1.10

i	*m*	*r*
–	3994	0
1	399	0*10+3994 Mod 10=0+4=4
2	39	4*10+399 Mod 10=40+9=49
3	3	49*10+39 Mod 10=490+9=499
4	0	499*10+3 Mod 10=4990+3=4993

операциями являются: `r:=10*r+m` **Mod** `10` (добавление очередной цифры к числу-«перевертышу») и `m:=m` **Div** `10` (изменение проверяемого числа). Процедура трассировки приведена в табл. 1.10. После ее выполнения написание программы — «техническая работа».

```
Program My6_1;
 Var n,m,r,i:Integer;
 Begin
  WriteLn('введите целое число, не большее 9999');
  ReadLn(n);
  m:=n;r:=0;
  For i:=1 To 4 Do Begin {так как число
                          четырехзначное}
   r:=r*10+m Mod 10;
   m:=m Div 10;
  End;
  If r=n Then WriteLn('ДА') Else WriteLn('НЕТ');
  ReadLn;
 End.
```

Измените программу так, чтобы была возможность обрабатывать целые числа из диапазона LongInt.

2. Даны натуральные четырехзначные числа n, k (n, $k \le 9999$). Из чисел от n до k выбрать те, запись которых содержит ровно три одинаковых цифры. Например, числа 6766, 5444, 0006, 0060 содержат ровно три одинаковых цифры.

Таблица 1.11

Номер цифры	1	2	3	4	
1	×	×	×		первое условие
2	×	×		×	второе условие
3	×		×	×	третье условие
4		×	×	×	четвертое условие

Если данное число содержит ровно три одинаковых цифры, то только одна из цифр отличается от остальных, то есть возможны всего четыре случая их расположения, приведенных в табл. 1.11. Пусть в качестве n и k введены числа 3732 и 3740. В переменных $a1$, $a2$, $a3$, $a4$ храним значения цифр текущего числа i (табл. 1.12).

Таблица 1.12

i	*a*1	*a*2	*a*3	*a*4	Результат сравнения
3732	3	7	3	2	ложь
3733	3	7	3	3	истина
3734	3	7	3	4	ложь
3735	3	7	3	5	ложь
3736	3	7	3	6	ложь
3737	3	7	3	7	ложь
3738	3	7	3	8	ложь
3739	3	7	3	9	ложь
3740	3	7	4	0	ложь

Примечание

Пусть вас не смущает «элементарность» выполняемых действий. Трудолюбие — одно из качеств «великого программиста»[6].

```
Program My6_2;
 Var n,k,i,a1,a2,a3,a4,m:Integer;
 Begin
  WriteLn(' Введите два числа, не больших 9999');
  ReadLn(n, k);
  For i:=n To k Do Begin
   m:=i; {выделение цифр: a1 - первая,
          a2 - вторая,a3 - третья, a4 - четвертая}
   a4:=m Mod 10; m:=m Div 10;
   a3:=m Mod 10; m:=m Div 10;
   a2:=m Mod 10;
   a1:=m Div 10;
   If ((a1=a2) And (a1=a3) And (a1<>a4)) Or
                                  {первое условие}
    ((a1=a2) And (a1=a4) And (a1<>a3)) Or
                                  {второе условие}
    ((a1=a3) And (a1=a4) And (a1<>a2)) Or
                                  {третье условие}
    ((a2=a3) And (a2=a4) And (a2<>a1))
                                {четвертое условие}
```

[6] А. Н. Венц в своей книге (Профессия — программист. Ростов н/Д: изд-во «Феникс», 1999.) приводит формулу великого программиста (ВП), выведенную экспериментальным путем: ВП=50% К+30% Т+10% О+5% З+5% ТЛ, где К — знать, *как* это делать, Т — трудолюбие, О — опыт, З — знание, ТЛ — талант.

```
    Then WriteLn(i:5);
  End;
  ReadLn;
End.
```

Задания для самостоятельной работы

Примечание

Перед написанием задаваемых ниже программ требуется выполнить ручную трассировку основных фрагментов решения; естественно, что циклы трассируются не при всех значениях управляющей переменной.

1. Напишите программу поиска всех двузначных чисел, в которых есть цифра n или само число делится на n.
2. Напишите программу возведения натурального числа в квадрат, используя следующую закономерность:
   $1^2=1$
   $2^2=1+3$
   $3^2=1+3+5$
   $4^2=1+3+5+7$
   ...
   $n^2=1+3+5+7+9+...+2n-1$
3. Напишите программу определения количества трехзначных натуральных чисел, сумма цифр которых равна заданному числу n.
4. Напишите программу вычисления суммы кубов чисел от 25 до 55.
5. Напишите программу поиска среди двузначных чисел таких, сумма квадратов цифр которых делится на 13.
6. Напишите программу поиска двузначных чисел, удовлетворяющих следующему условию: если к сумме цифр такого числа прибавить квадрат этой суммы, то получится само это число.
7. Напишите программу поиска трехзначных чисел, квадрат которых оканчивается тремя цифрами, составляющими исходное число.
8. Напишите программу поиска четырехзначного числа, которое при делении на 133 дает в остатке 125, а при делении на 134 дает в остатке 111.
9. Напишите программу поиска суммы положительных нечетных чисел, меньших 100.

10. Напишите программу нахождения суммы целых положительных чисел из промежутка от a до b, кратных 4 (значения переменных a и b вводятся с клавиатуры).

11. Напишите программу нахождения суммы целых положительных чисел, больших 20, меньших 100, кратных 3 и заканчивающихся на 2, 4 или 8.

12. Напишите программу поиска трехзначного числа, удовлетворяющего следующему условию: в числе зачеркнули первую цифру слева, полученное двузначное число умножили на 7 и получили данное число.

13. Напишите программу поиска всех трехзначных чисел, делящихся на 7, сумма цифр которых также кратна 7.

14. Напишите программу поиска четырехзначных чисел, у которых все четыре цифры различны.

15. Напишите программу поиска двузначных чисел, сумма цифр которых равна n ($0 < n \le 18$) и которые делятся без остатка на число q.

16. Дано четырехзначное число n. Выбросьте из записи числа n цифры 0 и 5, если они там имеются, оставив прежним порядок остальных цифр. Например, из числа 1509 должно получиться 19.

17. Натуральное число из n цифр является числом Армстронга, если сумма его цифр, возведенных в n-ю степень, равна самому числу (например, $153=1^3+5^3+3^3$). Напишите программу поиска всех чисел Армстронга, состоящих из трех и четырех цифр.

18. Дана последовательность из 20 целых чисел. Напишите программу, которая определяет количество чисел в наиболее длинной подпоследовательности из подряд идущих нулей.

19. Дано натуральное число. Напишите программу поиска всех его делителей и их суммы.

20. Напишите программу вычисления значения выражения $y=((...(20^2-19^2)^2-18^2)^2-...-1^2)^2$.

Занятие № 7. Оператор цикла While

План занятия

1. Оператор цикла While.
2. Эксперименты с программами (определение количества цифр в числе; проверка «гипотезы Сиракуз»).
3. Выполнение самостоятельной работы.

Оператор цикла While

While является оператором цикла с предусловием. Он имеет следующий вид:

```
While <логическое выражение> Do <оператор>;
```

Оператор While содержит логическое выражение, значение которого (True или False) управляет повторным выполнением оператора (после служебного слова Do), который может быть и составным. Значение выражения вычисляется перед выполнением оператора. Если результат равен True, то оператор выполняется, при значении False — нет. Таким образом, если в начале логическое выражение имеет значение False, оператор после Do вообще не выполняется. В операторе (составном операторе) обязательно предусматривается изменение значений переменных, влияющих на значение логического выражения. При невыполнении этого условия получаем пример того, что называется «зацикливанием». Простейший пример бесконечного цикла:

```
While True Do <оператор> ;
```

Экспериментальный раздел занятия

1. Дано натуральное число n. Подсчитать количество цифр данного числа.

Особенность данной задачи в том, что количество цифр в числе n заранее неизвестно. Именно поэтому необходимо использовать оператор While. Использование For потребовало бы или введения дополнительных переменных, или искусственного выхода из цикла, а мы договорились, что это плохо — каждый фрагмент нашей логики должен иметь одну точку входа и одну точку выхода. Подсчет количества цифр начнем с последней цифры числа. Увеличим счетчик цифр (k) на единицу. Число (m) уменьшим в 10 раз, убирая тем самым из него последнюю цифру (подсчитанную). С получившимся

числом проделаем ту же последовательность действий и так далее, пока число не станет равным нулю.

Для этого примера и всех заданий данного занятия требуется выполнять ручную трассировку. Пусть введено число n=65 387; присвоим это значение переменной с именем m; значение счетчика числа цифр (k) равно 0. Выполним действия, описанные выше; их результат приведен в табл. 1.13. Итак, окончательное значение переменной k равно 5 — в заданном числе 5 цифр. После этого работаем с программой My7_1 по традиционной схеме: набор, компиляция, сохранение, запуск и проверка работы на конкретных примерах.

Таблица 1.13

k	*m*
0	65387
1	6538
2	653
3	65
4	6
5	0

```
Program My7_1;
 Var  m, n: LongInt;
      k: Integer;{счетчик числа цифр}
 Begin
  WriteLn(' Введите целое число');
  ReadLn(n);
  m:=n;
  k:=0;
  While m<>0 Do Begin
   Inc(k); {k:=k+1;}
   m:=m Div 10;
  End;
  WriteLn(' В числе ',n,'  -  ',k,'  цифр !');
  ReadLn;
 End.
```

Модифицируя программу My7_1, решите следующие задачи:

- найти сумму цифр заданного числа;
- найти первую цифру числа, например для 7265 это цифра 7;
- поменять порядок цифр числа на обратный. Например, было 12345, стало 54321;
- найти количество четных цифр числа;
- найти самую большую цифру числа;
- найти сумму цифр числа, больших 5;
- ответить на вопрос, сколько раз данная цифра встречается в числе.

Придумайте еще 5 задач (как минимум), решаемых по данной схеме.

2. В программе My7_2 реализован процесс, относительно которого выдвинута так называемая гипотеза Сиракуз. Это процесс последовательного преобразования натурального числа *n* в 1. Запустив программу, проверяем ее работу при нескольких значениях *n* и видим, что предусмотренный результат каждый раз достигается. Отсюда мы делаем предположение, что работа программы завершится, то есть результат будет получен при любом значении *n*. Однако с достоверностью это неизвестно, доказательство факта завершения работы программы (алгоритма) при любом значении *n* до сих пор никем не получено[7]. Другими словами, есть алгоритм (программа), но его конечность, завершаемость работы во времени — открытый вопрос, требующий своего обоснования. Мораль: проверка циклов типа While требует особо тщательной работы, ибо подобные циклы «потенциально бесконечны во времени».

```
Program My7_2;
 Var n:Integer;
 Begin
  WriteLn('Введите натуральное число:');
  ReadLn(n);
  Write(n);
  While n<>1 Do Begin
   If n Mod 2=0 Then n:=n Div 2
    Else n:=(3*n+1) Div 2;
```

[7] Ноден П., Китте К. Алгебраическая алгоритмика (с упражнениями и решениями). – М.: Мир, 1999.

```
    Write(' - ',n);
  End;
  ReadLn;
End.
```

Последовательно запуская программу, оцените среднюю длину получаемых цепочек чисел при изменении n от 2 до 20. Как избавиться от этой ручной работы по многократному запуску программы?

Рассматривая полученные цепочки чисел, нетрудно заметить, что фрагменты этих цепочек часто повторяются. Например, 8⇒4⇒2⇒1, 5⇒16⇒8⇒4⇒2⇒1. Как использовать этот факт при подсчете средней длины цепочек для чисел (n) из большого интервала, например типа Integer? Потребуется ли в этом случае что-либо изменять в приведенном фрагменте программы?

Вопросы и задания для самостоятельной работы

1. Дана последовательность операторов:

```
a:=1;  b:=1;
While  a+b<8  Do  Begin
   a:=a+1;
   b:=b+2;
  End;
s:=a+b
```

Сколько раз выполняется проверка логического выражения в операторе While?

Определите значения переменных a, b, и s после завершения этой последовательности операторов.

2. Определите значения переменных a и b после выполнения операторов:

```
a:=1;
b:=1;
While a<=3 Do a:=a+1;
b:=b+1
```

3. Определите значение переменной s после выполнения следующих операторов:

а)
```
s:=0;
i:=0;
```

```
    While i<5 Do Inc(i);
    s:=s+100 Div i;
б)  s:=0; i:=1;
    While i>1 Do Begin
      s:=s+100 Div i;
      Dec(i);
    End;
```

4. Дан фрагмент программы с ошибками (их не больше 5) вычисления факториала *f* числа *n*:

```
k:=1;
f:=0;
While k<n Do f=f*k
k:=k+1;
```

Найдите эти ошибки.

5. Найдите и исправьте ошибки в следующем фрагменте программы, определяющей для заданного натурального числа *n* число, записанное цифрами числа *n* в обратном порядке.

```
p:=n;
While p>=0 Do Begin
 a:=a+p Mod 10;
 p:=p Div 10
End;
```

Примечание

Задания 1–5 рекомендуется выполнять, используя режим ручной трассировки.

6. Напишите программу поиска минимального числа, большего 300, которое нацело делится на 19.

7. Напишите программу, которая приписывает по 1 в начало и в конец записи числа *n*. Например, было *n*=3456, стало *n*=134561.

8. Напишите программу, которая меняет местами первую и последнюю цифры числа. Например, из числа 8547 должно быть получено число 7548.

9. Напишите программу, которая приписывает к исходному числу *n* такое же число. Например, из числа 1903 должно быть получено число 19031903.

10. Напишите программу, которая определяет, является ли заданное число степенью 3.
11. Напишите программу, которая проверяет, является ли заданное натуральное число палиндромом, то есть таким, десятичная запись которого читается одинаково слева направо и справа налево.

    Примечание

    Задача отличается от ранее рассмотренных (для трех- и четырехзначных чисел) тем, что количество цифр в числе неизвестно, а из этого следует, что тип используемого цикла должен быть другим.

12. Напишите программу, которая определяет, является ли последовательность цифр натурального числа при просмотре их справа налево возрастающей последовательностью. Например, для числа 76431 ответ положительный, для чисел 6331, 9782 — отрицательный.
13. Напишите программу, которая для введенной последовательности целых ненулевых чисел (признак окончания ввода — ввод 0, количество чисел не меньше 2) определяет:
    - является ли последовательность возрастающей;
    - есть ли в ней хотя бы одна пара одинаковых соседних чисел;
    - является ли последовательность знакочередующейся (3, –2, 4, –5, 0 — «да»; 5, –4, –7, 8, 0 — «нет»).
14. Напишите программу определения того, сколько раз в натуральном числе встречается его максимальная цифра. Например, в числе 581088 — 3 раза, в числе 4537 — 1 раз.
15. Напишите программу проверки того, является ли разность максимальной и минимальной цифр числа четной.

Занятие № 8. Оператор цикла Repeat – Until

План занятия

1. Оператор цикла Repeat – Until.
2. Эксперименты с программами (определение простоты числа; нахождение наибольшего общего делителя двух чисел с помощью алгоритма Евклида).
3. Выполнение самостоятельной работы.

Оператор цикла Repeat – Until

Оператор Repeat (повторять) – Until (до тех пор, пока) содержит логическое выражение (после Until), которое управляет повторением выполнения последовательности операторов, записанных между Repeat и Until. Повторение продолжается до тех пор, пока логическое выражение не примет значение True. Последовательность операторов выполняется по меньшей мере один раз, ибо логическое выражение вычисляется после каждого выполнения данной последовательности.

Общий вид оператора:

```
Repeat
 <оператор 1>;
 <оператор 2>;
 ...
 <оператор n>;
Until <логическое выражение>;
```

При использовании оператора Repeat – Until (цикла с постусловием) необходимо учитывать следующее:

- перед первым выполнением оператора логическое выражение его окончания (или продолжения) должно быть определено;
- последовательность операторов должна содержать хотя бы один оператор, влияющий на значение логического выражения, иначе оператор Repeat – Until работает бесконечно долго;
- логическое выражение в конечном итоге должно принять значение True.

Пример простейшей, бесконечной по времени работы, конструкции: **Repeat** … **Until False**;

Экспериментальный раздел занятия

1. Целое положительное число p называется простым, если оно имеет только два делителя, а именно 1 и p. По соглашению 1 не считают простым числом. Начало последовательности простых чисел имеет вид:

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, ...

Научимся устанавливать факт, является ли число n простым. Ниже приведен текст решения. Считаем, что делители числа находятся в интервале от 2 до n Div 2, точнее, в интерва-

ле от 2 до целой части Sqrt(n), то есть целой части квадратного корня из числа n.

```
Program My8_1;
 Var i,n:LongInt;
 Begin
  WriteLn('Введите натуральное число');
  ReadLn(n);
  i:=1;
  Repeat
   i:=i+1;
  Until (i> n Div 2) Or (n Mod i=0);
  If i> n Div 2 Then Writeln('Число ',n,' простое')
   Else WriteLn('Число  ',i,' первый делитель
                числа ',n);
  ReadLn;
 End.
```

Естественно, что эту задачу можно решить и с использованием оператора While. Сделайте эту небольшую работу. А затем измените программу так, чтобы выполнялся вывод всех делителей числа n.

Подсказка

Логическое выражение оператора Repeat – Until упростится, останется только условие $i>n$ Div 2, а в операторах тела цикла появится:

```
If n Mod i=0 Then WriteLn(... ..., i)
```

Как известно, всякое натуральное число $n>1$ разлагается на произведения простых чисел (простые сомножители). Это разложение однозначно с точностью до порядка записи простых чисел в произведении.

Выберите интервал чисел, не очень большой, например от 101 до 120, и, используя программу, найдите разложения этих чисел на простые сомножители. Ваши записи должны иметь вид: $n=p_1^{\alpha 1}\cdot p_2^{\alpha 2}\cdot\ldots\cdot p_q^{\alpha q}$, где $(\alpha_i\geq 0)$, p_i — простые числа, например $168=2^3\cdot 3^1\cdot 5^0\cdot 7^1$.

2. Наибольший общий делитель (НОД) двух целых чисел a и b — это наибольшее целое число, которое делит нацело оба числа. Для нахождения НОД чисел a и b можно представить оба числа в виде: $a=p_1^{\alpha 1}\cdot p_2^{\alpha 2}\cdot\ldots\cdot p_q^{\alpha q}$ и $b=p_1^{\beta 1}\cdot p_2^{\beta 2}\cdot\ldots\cdot p_q^{\beta q}$, а затем найти НОД по формуле $(a,b)=p_1^{\min(\alpha 1\beta 1)}\cdot p_2^{\min(\alpha 2\beta 2)}\cdot\ldots\cdot p_q^{\min(\alpha q\beta q)}$, где некоторые степени простых чисел могут быть и нулевыми.

Отложим исследование этого метода нахождения НОД — у нас пока не хватает знаний. Обратимся к алгоритму греческого математика Евклида, он открыл его в 330–320 гг. до н. э. Алгоритм основан на следующем свойстве целых чисел. Пусть a и b — одновременно не равные нулю целые неотрицательные числа и $a \geq b$. Тогда если $b=0$, то НОД(a,b)=a, а если $b \neq 0$, то для чисел a, b и r, где r — остаток от деления a на b, выполняется равенство НОД(a,b)=НОД(b,r). Действительно, $r=a$ Mod $b=a-(a$ Div $b)\cdot b$. Если какое-то число делит нацело и a, и b, то из приведенного равенства следует, что оно делит нацело и числа r и b.

Например, пусть $a=48$, а $b=18$, найдем их наибольший общий делитель. Приведем ручную трассировку логики, она, как обычно, сведена в табл. 1.14.

Таблица 1.14

a	*b*		Результаты
48	18		
48 Mod 18=12	18	*a>b*	НОД(48,18)=НОД(12,18)
12	18 Mod 12=6	*a<b*	НОД(12,18)=НОД(12,6)
12 Mod 6=0	6	*a>b*	НОД(12,6)=НОД(0,6)
0	6	*a*=0	НОД(0,6)=6

Таким образом, НОД(48,18)=6.

Программная реализация алгоритма Евклида с использованием оператора Repeat – Until имеет вид:

```
Program My8_2;
 Var a, b: LongInt;
 Begin
  WriteLn('Введите два числа ≠ 0');
  ReadLn(a,b);
  Repeat
   If a>b Then a:=a Mod b Else b:=b Mod a;
  Until (a=0) Or (b=0);
  WriteLn('НОД=',a+b);
  ReadLn;
 End.
```

Измените программу так, чтобы вместо оператора Repeat – Until использовался оператор While. Какое ограничение в этом случае мы можем убрать?

Последовательность чисел

$e_1=1+1=2$,

$e_2=2+1=3$,

$e_3=2\cdot 3+1=7$,

$e_4=2\cdot 3\cdot 7+1=43$,

$e_5=2\cdot 3\cdot 7\cdot 43+1=1807$,

$e_6=2\cdot 3\cdot 7\cdot 43\cdot 1807+1=3263443$,

$e_7=2\cdot 3\cdot 7\cdot 43\cdot 1807\cdot 3263443+1$ и так далее называют числами Евклида. Первые 4 числа наталкивают на мысль о том, что евклидовы числа простые. Однако уже e_5 является составным — $1807=13\cdot 139$. Известно, что евклидовы числа взаимно простые, то есть НОД(e_i,e_j)=1 при $i\neq j$. Проверьте этот факт с помощью программы My8_2 для первых 6 чисел Евклида. Для работы с остальными числами Евклида типа LongInt недостаточно, необходимо изучить по крайней мере основы «длинной» арифметики[8] или использовать другую систему программирования. С помощью программы My8_1 покажите, что число e_6 простое.

Вопросы и задания для самостоятельной работы

1. Числа вида 2^p-1, где p — простое число, называются числами Мерсенна (1588–1648). Являются ли числа Мерсенна при значениях p=2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31 простыми? Для ответа на этот вопрос напишите соответствующую программу. Программа должна состоять из двух частей: в первой вычисляется число Мерсенна для значения p (вводится с клавиатуры), во второй проверяется, является ли оно простым.
2. Линейные уравнения с двумя переменными (диофантовы уравнения), то есть уравнения вида: $a\cdot x+b\cdot y=c$, где a и b не равны нулю одновременно, имеют целые решения тогда и только тогда, когда d (d=НОД(a,b)) делит нацело значение c. Причем если x_0, y_0 — частное решение, то все решения имеют вид $x=x_0-n\cdot(b \text{ Div } d)$, $y=y_0+n\cdot(a \text{ Div } d)$ для всех n.

[8] Окулов С. М. Программирование в алгоритмах. — М.: БИНОМ. Лаборатория знаний, 2002.

Пример:

$12 \cdot x - 30 \cdot y = 84$, НОД(12, –30)=6, 84 делится нацело на 6. Уравнение разрешимо. Одним из его решений является $(x, y)=(2, -2)$. Все остальные решения имеют вид $x=2+5 \cdot n$, $y=-2+2 \cdot n$.

Даны целые числа a, b, c. Напишите программу определения разрешимости соответствующего диофантова уравнения, и если оно разрешимо — поиска частного решения.

3. Пусть a и b — ненулевые целые числа. Целое число $m>0$ называется наименьшим общим кратным (НОК) чисел a и b, если m делится и на a, и на b нацело, а также для любого c, которое делится нацело и на a и на b, верно, что оно делится нацело и на m.

Если a и b — ненулевые числа, то их наименьшее общее кратное существует и справедливо равенство НОК(a, b)=Abs($a \cdot b$) Div НОД(a, b). Напишите программу нахождения НОК двух ненулевых чисел.

4. Числа Фибоначчи (f_n) определяются формулами $f_0=f_1=1$ и $f_n=f_{n-1}+f_{n-2}$ при $n=2, 3, \ldots$, то есть это бесконечная последовательность чисел вида 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... Напишите программу:
   - определения номера последнего числа Фибоначчи, которое входит в диапазон типа Integer (LongInt);
   - вычисления s — суммы первых чисел Фибоначчи, таких, что значение s не превышает диапазона типа Integer (LongInt).
5. Совершенным числом называется число, равное сумме всех своих делителей, меньших, чем оно само. Например: 6=1+2+3, 28=1+2+4+7+14. Древним грекам были известны только четыре первых совершенных числа. Напишите программу, проверяющую, является ли заданное натуральное число совершенным.

   Примечание

   Евклидом доказано, что каждое число вида $2^{p-1} \cdot (2^p-1)$ является совершенным числом, если 2^p-1 — простое число. Л. Эйлер показал, что все четные совершенные числа находятся по формуле Евклида, а относительно нечетных совершенных чисел ничего неизвестно до сих пор.
6. Автоморфным называется такое число, которое равно последним цифрам своего квадрата. Например: $5^2=25$,

25^2=625. Очевидно, что автоморфные числа должны оканчиваться либо на 1, либо на 5, либо на 6. Напишите программу нахождения автоморфных чисел (с учетом приведенного факта), не превышающих значения 999.

Примечание

Не забывайте о диапазонах переменных целого типа, 999^2=998001.

7. Кубические автоморфные числа равны последним цифрам своих кубов. Например: 6^3=216. Верно ли, что и такие числа должны оканчиваться либо на 1, либо на 5, либо на 6? Напишите программу (с учетом этого факта) нахождения двузначных, трехзначных кубических автоморфных чисел.

Справочные сведения

Ошибки в программах разделяют по типам на синтаксические, семантические и логические. С первыми мы встречались на предыдущих занятиях. Они достаточно просто устраняются и говорят о несоответствии текста программы правилам языка Паскаль. Вторые возникают на стадии выполнения программы. Например, деление на ноль не устраняется на стадии компиляции, а приводит к ошибке на стадии выполнения, ибо зависит от конкретного значения переменной. Третий тип ошибок — самый сложный в обнаружении. Программа работает, но работает так, как написана, а не так, как требуется. Такие случаи являются следствием многих причин. Главное, что их сложно обнаружить. В системе Паскаль имеется отладчик, позволяющий находить в программах ошибки третьего типа. Он работает с исходным текстом программы, позволяет выполнять программу по шагам и отслеживать изменение переменных программы в процессе работы последней.

Перечислим основные команды отладчика и соответствующие им клавиши, позволяющие выполнять программу по шагам и отслеживать значения переменных (табл. 1.15). Шагом при отладке является строка программы. Так, если в строке исходного текста программы записано несколько операторов, то они будут выполнены за один шаг.

Таблица 1.15

Команда меню	Функциональная клавиша	Назначение
Run/Run	Ctrl+F9	Запускает программу
Run/Go to cursor	F4	Выполняет программу до строки, в которой находится курсор
Run/Trace into	F7	Выполняет оператор (операторы), записанный в текущей строке. Если в этой строке записан вызов процедуры или функции, то выполняется вход в эту процедуру или функцию и отслеживается их работа
Run/Step over	F8	Выполняет оператор, записанный в текущей строке без вхождения в процедуры или функции
Debug/Watch		Открывает окно для наблюдения за значениями переменных (окно Watches)
Debug/ Breakpoints		Открывает окно для работы с точками останова
Debug/Evaluate...	Ctrl+F4	Открывает окно Evaluate and Modify. В нем набирается выражение (можно с использованием переменных программы) и вычисляется его значение
Debug/Add watch	Ctrl+F7	Открывает окно Add watch для набора отслеживаемых значений выражений или переменных. Эти действия можно выполнить и другим способом: сделать окно Watch активным и использовать клавиши Insert и Delete для вставки и удаления выражений и переменных
Debug/ Add breakpoint	Ctrl+F8	Текущая строка (в ней находится курсор) становится точкой останова программы

Занятие № 9. Вложенные циклы

План занятия

1. Вложенные циклы.
2. Эксперименты с программами (вычисление выражения $1^k+2^k+\ldots+n^k$; нахождение цифрового корня числа; решение старинной задачи о быках, коровах и телятах; нахождение натуральных чисел, удовлетворяющих определенному условию; сведение чисел, кратных 3, к числу 153).
3. Выполнение самостоятельной работы.

Вложенные циклы

Для решения задачи достаточно часто требуется использовать две и более циклических конструкций, одна из которых помещается в тело цикла другой. Такие конструкции называют вложенными циклами. Как внутренний, так и внешний циклы могут быть любыми из трех рассмотренных ранее видов. Правила организации внешнего и внутреннего циклов такие же, как и соответствующих простых операторов. Однако при использовании вложенных циклов необходимо соблюдать следующее условие: внутренний цикл должен полностью укладываться в циклическую часть внешнего цикла.

Экспериментальный раздел занятия

1. Даны натуральные числа n и k. Составить программу вычисления выражения $1^k+2^k+...+n^k$.

Для вычисления указанной суммы целесообразно использовать оператор For с управляющей переменной i, изменяющейся от 1 до n. В теле цикла, во-первых, вычисляется очередное значение $y=i^k$ и, во-вторых, происходит накопление суммы прибавлением полученного слагаемого к сумме всех предшествующих (s:=s+y).

```
Program My9_1;
 Var n, k, y, i, s, j: Integer;
 Begin
  WriteLn('Введите исходные данные n  и k ');
  ReadLn(n,k);
  s:=0;
  For i:=1 To n Do Begin
   y:=1;
   For j:=1 To k Do y:=y*i;
   s:=s+y;
  End;
  WriteLn('Ответ: ',s);
 End.
```

А сейчас попробуем работать с отладчиком системы программирования; его команды были приведены в табл. 1.15.

- Выполните команду **Debug/Watch**. Появится окно **Watches**.
- Используя клавишу Ins (инициируется открытие окна **Add Watch**), введите переменные программы (n, k, s, i, j, y).

- Измените положение и размер окна **Watches** на экране. Оно должно находиться в правом верхнем углу и быть таким, чтобы все введенные переменные были обозримы. С этой целью выполните команду **Window/Size/Move** (Ctrl+F5) и используйте клавиши Shift+(⇐, ⇑) для изменения размера окна и клавиши (⇒, ⇓) для его перемещения.
- Выполните команду **Run/Step over** (F8). Строка с оператором Begin выделяется другим цветом (курсор). При нажатии на F8 курсор перемещается на следующую строку — пошаговое выполнение программы.
- Выполните программу в пошаговом режиме (последовательное нажатие на F8), отслеживая изменение значений переменных в окне **Watches**. При выполнении **ReadLn**(n,k) введите, например, значения 4 и 2 для того, чтобы количество повторений в операторах For было не очень большим.

Продолжим изучение отладчика.

- Установите курсор на строку программы с оператором s:=s+y. Выполните команду **Debug/Add Breakpoint** (Ctrl+F8). Строка будет выделена другим цветом. Таким образом в программе создается точка останова.
- Запустите программу (Ctrl+F9). Работа программы приостанавливается в том случае, когда достигнута точка останова.
- Выполните один шаг в работе программы — нажмите клавишу F8.
- Выполните команду **Debug/Evaluate...** В строке **Expression** окна **Evaluate and Modify** наберите имя переменной *s*. В строке **Result** окна мы видим значение переменной *s*, оно равно 1.
- Продолжите работу с программой по описанной схеме. Следующее значение переменной *s* равно 5.

Методические рекомендации

Рекомендуется максимально использовать отладчик системы программирования при выполнении заданий как этого, так и следующих занятий. И так до тех пор, пока не появятся устойчивые навыки работы с ним.

Проведите эксперименты с программой:

- Измените программу так, чтобы вычислялась сумма $1^1+2^2+\ldots+n^n$.
- Пусть значение k зафиксировано, например равно 5. Определить значение n, при котором диапазон целого типа данных Integer окажется недостаточным для хранения суммы степеней чисел.

2. Сложим все цифры какого-либо числа. Получим новое число, равное сумме всех цифр исходного числа. Продолжим этот процесс до тех пор, пока не получим однозначное число (цифру), называемое цифровым корнем данного числа. Например, цифровой корень числа 34 697 равен 2 (3+4+6+9+7=29; 2+9=11; 1+1=2). С помощью отладчика проверьте работоспособность программы нахождения цифрового корня натурального числа.

```
Program My9_2;
 Var n,k,s:LongInt;
 Begin
  WriteLn('Введите число');
  ReadLn(n);
  s:=n;
  While s>9 Do Begin
    k:=s;
   s:=0;
   Repeat
    s:=s+k Mod 10;
    k:=k Div 10;
   Until k=0;
  End;
  WriteLn('цифровой корень числа ',n,' равен ',s);
 End.
```

3. *Старинная задача*. Сколько нужно купить быков, коров и телят на 100 рублей, если требуется купить в сумме 100 голов скота, причем плата за быка — 10 рублей, за корову — 5 рублей, за теленка — полтинник (0,5 рубля)?

Решение:

Обозначим через b количество быков; k — количество коров; t — количество телят. После этого можно записать два уравнения: $10b+5k+0{,}5t=100$ и $b+k+t=100$. Преобразуем первое уравнение: $20b+10k+t=200$.

На 100 рублей можно купить:

- не более 10 быков, то есть $0 \le b \le 10$;
- не более 20 коров, то есть $0 \le k \le 20$;
- не более 200 телят, то есть $0 \le t \le 200$.

Таким образом, получаем:

```
Program My9_3;
 Var b, k, t: Integer;
 Begin
  For b:=0 To 10 Do
   For k:=0 To 20 Do
    For t:=0 To 200 Do
     If (20*b+10*k+t=200) And (b+k+t=100) Then
      WriteLn('быков ',b,' коров ',k,' телят ',t);
 End.
```

Сколько раз будет проверяться условие в данной программе (оператор If)? Значение переменной b изменяется 11 раз (от 0 до 10), для каждого ее значения переменная k изменяется 21 раз, а для каждого значения переменной k переменная t изменяется 201 раз. Таким образом, условие будет проверяться $11 \cdot 21 \cdot 201 = 46\,431$ раз. Но если известно количество быков и коров, то количество телят можно вычислить по формуле $t = 100 - (b + k)$ и цикл по переменной t исключается.

```
Program My9_3m;
 Var b, k, t: Integer;
 Begin
  For b:=0 To 10 Do
   For k:=0 To 20 Do Begin
    t:=100-(b+k);
    If (20*b+10*k+t=200) Then
     WriteLn('быков ',b,' коров ',k,' телят ',t);
   End;
 End.
```

При этом решении условие проверяется $11 \cdot 21 = 231$ раз. Попробуйте еще уменьшить количество проверок.

4. Напишите программу, которая находит все четырехзначные числа $abcd$ (a, b, c, d — цифры числа, причем все они различны), для которых выполняется условие: $ab - cd = a + b + c + d$. Другими словами, разность чисел, составленных из старших цифр числа и из младших, равна сумме цифр числа.

Задачу можно решать разными способами. Один из них — перебор всех четырехзначных чисел и проверка выполнения условия. Попробуем сократить перебор, для этого преобразуем условие. Из равенства $10 \cdot a+b-(10 \cdot c+d)=a+b+c+d$ получаем $9 \cdot (a-c)=2 \cdot (c+d)$ или $(a-c)/(c+d)=2/9$, $a=c+2$, $d=9-c$ и $0 \le c \le 7$.

```
Program My9_4;
 Var a,b,c,d:Integer;
 Begin
  For c:=0 To 7 Do Begin
   a:=c+2;
   d:=9-c;
   For b:=0 To 9 Do
    If (b<>c)And(b<>a)And(b<>d) Then
     WriteLn(a,b,c,d);
  End;
 End.
```

5. Дано натуральное число, кратное 3. Найдем сумму кубов цифр данного числа. Получим новое число. Применим к нему такое же преобразование.Какой вид будет иметь полученная последовательность чисел? Оказывается, при любом исходном числе в ней рано или поздно появляется число 153, которое затем не меняется.

Известно, что если число n, состоящее из k цифр, то есть $a_1a_2...a_k$, делится на 3, то и сумма его цифр $s=a_1+a_2+...+a_k$ делится на 3. Это необходимый и достаточный признак делимости числа на 3. Куб этой суммы $s^3=a_1^3+a_2^3+...+a_k^3+3 \cdot P$, где P — сумма попарных произведений цифр и их квадратов. Отсюда следует, что $a_1^3+a_2^3+...+a_k^3=s^3-3 \cdot P$, то есть сумма кубов цифр числа n также кратна 3. Но все такие последовательности являются сходящимися к числу 153 — числу, равному сумме кубов своих цифр. Это доказывается методом математической индукции.

```
Program My9_5;
 Var n,m,t,s,q:LongInt;
 Begin
  WriteLn('Введите число, оно умножается на 3,
           и для полученного числа строится
           последовательность');
  ReadLn(n);
  m:=3*n;
```

```
  Write(m,' ');
  Repeat
   s:=0;t:=m;
   While m>0 Do Begin
    q:=m Mod 10;
    s:=s+q*q*q;
    m:=m Div 10;
   End;
   m:=s;
   Write(m,' ');
  Until m=t;
  ReadLn;
 End.
```

Проверьте, что эта логика работает для всех чисел, например меньших 33 333. Для этого потребуется, естественно, дополнить программу.

Вопросы и задания для самостоятельной работы

1. Что будет выведено на экран монитора после выполнения следующего фрагмента программы при n=6?

```
a:=1;
b:=1;
For i:=0 To n Do Begin
 For j:=1 To b Do Write('*');
 WriteLn;
 c:=a+b; a:=b; b:=c;
End;
```

Какая задача решается в этом фрагменте программы?

2. Что будет выведено на экране монитора после выполнения следующего фрагмента программы при a=13 305?

```
b:=0;
While a<>0 Do Begin
 b:=b*10+a Mod 10;
 a:=a Div 10;
End;
Write(b);
```

Решение какой задачи выражает этот фрагмент программы?

3. Дано натуральное число q. Напишите программу для нахождения всех прямоугольников, площадь которых равна q и стороны выражены натуральными числами.
4. Напишите программу для графического изображения делимости чисел от 1 до n (n — исходное данное). В каждой строке надо печатать число и столько плюсов, сколько делителей у этого числа. Например, если исходное данное — число 4, то на экране должно быть напечатано:

```
1+
2++
3++
4+++
```

5. Дано натуральное число n. Требуется выяснить, можно ли представить его в виде суммы трех квадратов натуральных чисел. Если можно, то:
   - указать тройку x, y, z таких натуральных чисел, что $x^2+y^2+z^2=n$;
   - указать все тройки x, y, z таких натуральных чисел, что $x^2+y^2+z^2=n$.
6. Напишите программу поиска натурального числа от 1 до 10 000 с максимальной суммой делителей.
7. Даны натуральные числа a, b ($a<b$). Напишите программу нахождения всех простых чисел p, удовлетворяющих неравенству $a\le p\le b$.
8. Даны натуральные числа n, m. Напишите программу поиска всех натуральных чисел, меньших n, квадрат суммы цифр которых равен m.
9. Даны натуральные числа n и m. Напишите программу поиска всех пар дружественных чисел, лежащих в диапазоне от n до m. Два числа называются дружественными, если каждое из них равно сумме всех делителей другого (сами числа в указанные суммы не входят).
10. Напишите программу, которая переставляет цифры натурального числа таким образом, чтобы образовалось наименьшее число, записанное этими же цифрами.
11. Напишите программу, выводящую k-ю цифру последовательности:
    - 12345678910111 2..., в которой выписаны подряд все натуральные числа;

- 14916253649..., в которой выписаны подряд квадраты всех натуральных чисел;
- 1123581321..., в которой выписаны подряд все числа Фибоначчи.

12. Напишите программу возведения заданного числа в третью степень с использованием следующей закономерности:
$1^3=1$
$2^3=3+5$
$3^3=7+9+11$
$4^3=13+15+17+19$
$5^3=21+23+25+27+29$...

13. Напишите программу нахождения всех натуральных чисел m, n, k из интервала [1, 10], удовлетворяющих уравнению $n^2+m^2=k^2$.

Примечание
Решения, которые получаются перестановкой n и m, считать совпадающими.

14. Напишите фрагмент программы для решения указанной ниже задачи и обоснуйте, почему был выбран тот или иной вариант оператора цикла:

- вычислить факториал некоторого числа p;
- определить, является ли заданное число p факториалом какого-либо числа, и если да, то найти это число;
- определить, является ли заданное число степенью числа 3;
- вычислить $y=1!+2!+3!+\ldots+n!$
- вычислить x^n, где n — целое положительное число;
- вычислить значения x^1, x^2, x^3,..., x^n.

15. Дано произвольное трехзначное число n. Если записать все трехзначные числа, состоящие из тех же цифр, что и n, и найти их среднее арифметическое, получим некоторое число s_n. Напишите программу поиска всех n, для которых s_n состоит из тех же цифр, что и само n.

16. Стороны прямоугольника заданы натуральными числами m и n. Напишите программу поиска количества квадратов (чьи стороны выражены натуральными числами), которые можно отрезать от данного прямоугольника, если каждый раз отрезается квадрат максимально большой площади.

17. Даны натуральные числа n и p. Напишите программу поиска всех натуральных чисел, меньших n и взаимно простых с p.

18. Даны целые числа p и q. Напишите программу поиска всех делителей числа q, взаимно простых с p.

19. Сумма квадратов длин катетов a и b прямоугольного треугольника равна квадрату длины гипотенузы c: $a^2+b^2=c^2$. Тройка натуральных чисел, удовлетворяющих этому равенству, называется пифагоровыми числами. Напишите программу нахождения нескольких наименьших троек пифагоровых чисел, используя следующие формулы:

$a=u \cdot v$;
$b=(u \cdot u-v \cdot v)$ Div 2;
$c=(u \cdot u+v \cdot v)$ Div 2,
где u и v — взаимно простые нечетные натуральные числа и $u>v$.

20. Напишите программу поиска наименьшего натурального числа n, представимого двумя различными способами в виде суммы кубов двух натуральных чисел x^3 и y^3 ($x \geq y$).

21. Даны натуральные числа m, n_1, n_2,..., n_m ($m \geq 2$). Напишите программу вычисления наибольшего общего делителя этих чисел — НОД(n_1, n_2, ..., n_m), воспользовавшись соотношением НОД(n_1, n_2, ..., n_m)=НОД(НОД(n_1, n_2, ..., n_{m-1}), n_m) и алгоритмом Евклида.

22. Напишите программу поиска всех простых несократимых дробей, заключенных между 0 и 1, знаменатели которых не превышают 7 (дробь задается двумя натуральными числами — числителем и знаменателем).

1.2. Процедуры и функции – элементы структуризации программ

Занятие № 10. Одномерные массивы. Работа с элементами

План занятия

1. Описания типов данных.
2. Одномерные массивы.
3. Эксперименты с программами (поиск суммы элементов массива, кратных заданному числу; нахождение элементов с определенными свойствами в массиве целых чисел; формирование значений элементов массива с помощью генератора случайных чисел; вычисление факториала числа).
4. Выполнение самостоятельной работы.

Описание типов данных

До этого занятия в наших программах использовались только два типа данных: Integer и Boolean. Тип переменной определяет множество значений, которые она может принимать, и операции, которые могут быть над ней выполнены. С каждой встречающейся в программе переменной может быть связан только один тип данных. Перечнем Integer и Boolean не исчерпывается весь список типов данных в Паскале (см. хотя бы табл. 1.1). Он не исчерпывается любым списком, т. к. в языке программирования есть возможность для конструирования новых типов данных. Типы Integer и Boolean относятся к простым (простые — потому что не могут состоять ни из каких других типов). Но так или иначе, любой, самый сложный тип данных, который вы создали в своей программе, сводится к простым типам или строится на их основе.

Описание типов данных имеет следующий вид:

```
Type <имя_типа>=<тип_данных>;
```

После этого в разделе описаний переменных у вас появляется возможность ссылаться на введенный тип данных — он ведь имеет имя:

```
Var <имя_переменной>:<имя_типа>;
```

Простой пример:

Необходимо найти сумму 5 целых чисел. Решение очевидно.

```
Program My10_1;
 Var  a1,a2,a3,a4,a5,s: Integer;
 Begin
  WriteLn('Введите пять целых чисел ');
  ReadLn(a1,a2,a3,a4,a5);
  s:=a1+a2+a3+a4+a5;
  WriteLn('Их сумма равна ',s)
  ReadLn;
 End.
```

А если требуется найти сумму 30 целых чисел? Решение по аналогии требует введения 30 однотипных переменных, то есть окажется весьма громоздким.

Одномерный массив

Одномерный массив — это фиксированное количество элементов одного и того же типа, объединенных одним именем, где каждый элемент имеет свой номер. Обращение к элементам массива осуществляется с помощью указания имени массива и номеров элементов. Нам для работы требуется массив из 30 целых чисел.

Опишем в разделе типов нужный тип — одномерный массив, состоящий из 30 целых чисел:

```
Type MyArray=Array [1..30] Of Integer;
```

Напомним, что раздел типов начинается со служебного слова Type, после этого идет имя нового типа и его описание. Между именем типа и его описанием ставится знак «равно» (в разделе переменных между именем переменной и ее описанием ставится двоеточие). В нашем случае MyArray — имя нового типа данных; Array — служебное слово (в переводе с английского означает «массив», «набор»); далее следует запись [1..30] — в квадратных скобках указывается номер первого элемента, затем, после двух точек, — номер последнего элемента массива; Of — служебное слово (в переводе с английского «из»); Integer — тип элементов массива. И решение, простое решение без 30 переменных, выглядит так:

```
Program My10_1m;
 Const n=30;
 Type MyArray=Array[1..n]  Of  Integer;
 Var  A: MyArray;
      s,i: Integer;
 Begin
  WriteLn('Введите ',n, ' чисел');
  For i:=1 To n Do ReadLn(A[i]);
  s:=0;
  For  i:=1 To n Do s:=s+A[i];
  WriteLn('Их сумма равна ',s);
  ReadLn;
 End.
```

Экспериментальный раздел занятия

1. Найдите сумму элементов массива из My10_1m, кратных заданному числу. Предыдущее решение изменится незначительным образом. Добавляется описание еще одной переменной для хранения значения числа, на кратность которому проверяются значения элементов массива. Появляются операторы:

```
WriteLn('введите число ');
ReadLn(k);
```

и изменяется оператор из тела цикла:

```
If A[i] Mod k=0 Then s:=s+A[i];
```

Примечание

Не забудьте изменить значение константы *n*. Вводить при каждом запуске программы 30 чисел — утомительное занятие.

Измените программу так, чтобы определялось количество положительных и отрицательных элементов в данном массиве. Суть основного изменения программы заключается во введении двух переменных (счетчиков — *pos*, *neg*) для хранения значений количества положительных и отрицательных элементов в массиве. Найдите место в программе для следующих строк программного кода:

```
pos, neg:Integer;
pos:=0;
neg:=0;
If A[i]>0 Then pos:=pos+1
```

```
Else If A[i]<0 Then neg:=neg+1;
WriteLn(pos:4,neg:4);
```

Данная программа не решает задачу подсчета количества нулевых элементов в массиве. Модифицируйте программу для решения и этой задачи.

Измените программу так, чтобы номера четных элементов массива *A* записывались в массив *B*. Введение массива *B* и работа с ним требуют введения переменной (*j*) для обращения к элементам *B*. Суть решения заключается в просмотре элементов массива *A*, выявлении четных элементов и записи их номеров по текущему значению переменной *j* в массив *B*.

Примечание

Мы записываем не значения элементов, а их номера! К сожалению, различие между этими понятиями осознается учащимися после первого знакомства не сразу, решения одной задачи недостаточно...

```
Program My10_2;
 Const n=10;
 Type MyArray=Array[1..n]  Of  Integer;
 Var  A,B: MyArray;
      i,j: Integer;
 Begin
  WriteLn('Введите ',n, ' чисел ');
  For i:=1 To n Do ReadLn(A[i]);
  j:=0;
  For  i:=1 To n Do
   If  A[i] Mod 2=0 Then Begin
    j:=j+1;
    B[j]:=i;
   End;
  For i:=1 To j Do Write(B[i]:3);
  WriteLn;
  ReadLn;
 End.
```

2. Научимся определять, есть ли в массиве элемент с заданными свойствами, например есть ли отрицательный элемент. А если таких элементов несколько? А если требуется определять номера этих элементов в массиве или выводить их значения? Это все разные задачи. Уточним условие — ищем номер последнего отрицательного элемента.

Напрашивается очевидное решение, его фрагмент выглядит следующим образом:

```
k:=0;
For i:=1 To n Do
 If A[i]<0 Then k:=i;
If k=0 Then <отрицательных элементов в массиве
            нет>
Else <последним отрицательным элементом является
      i-й по счету в массиве A>;
```

Примечание

В угловых скобках указываются описания тех фрагментов программы, которые при реализации следует записать на языке Паскаль.

Это решение плохое. Действительно, если единственный отрицательный элемент в массиве записан на первом месте, то мы выполняем $n-1$ лишнее сравнение, просматривая массив до конца. Договоримся о том, что наш программный код должен экономно, рационально использовать ресурсы компьютера, в частности и время его работы. Эффективный алгоритм значит в информатике больше, чем сверхбыстродействующий компьютер.

Следующее изменение:

```
If A[i]<0 Then Begin  k:=i; Break; End;
```

решает возникший вопрос, но... Это изменение приводит к тому, что наш небольшой фрагмент логики имеет одну точку входа и две точки выхода. Обилие Break, GoTo превратит программу в «спагетти»[9], а у нас не 60-е годы, первый виток развития технологий программирования мы оставили истории.

И еще одно замечание. В элементарной задаче задействованы две переменные. А если задача не элементарная и мы будем так же их «транжирить»? Нам не хватит, образно выражаясь, никакого алфавита. Приемлемый вариант решения имеет вид:

```
i:=n;
While (i>=1) And (A[i]>0) Do i:=i-1;
If i<1 Then <отрицательных элементов нет
             в массиве>
```

[9] «Спагетти» – образ программы, в которой преобладают операторы передачи управления типа GoTo, Break, Exit, Continue.

```
Else <последний отрицательный элемент имеет
      номер i>;
```

Еще немного изменим задачу: найдем значения и номера всех отрицательных элементов. В этом случае использование оператора For неизбежно, т. к. нам необходимо просмотреть все элементы массива. Итак, не наше желание определяет тип используемого оператора цикла, а решаемая задача.

```
{$R+}
Program My10_3;
 Const n=7;
 Type MyArray=Array[1..n] Of Integer;
 Var A:MyArray;
    i:Integer;
 Begin
  WriteLn('Ввод элементов массива. Не забудьте
           об отрицательных элементах. ');
  For i:=1 To n Do Read(A[i]);
  WriteLn('Вывод отрицательных элементов массива
           и их номеров (индексов)');
  For i:=1 To n Do
   If A[i]<0 Then WriteLn(A[i]:5,i:5);
  ReadLn;
 End.
```

Директивы компилятора управляют режимом компиляции. Директива начинается с символа $ после открывающей скобки комментария, за которым следует имя директивы (состоящее из одной или нескольких букв), определяющее ее назначение. Контроль вхождения в диапазон осуществляется с помощью директивы {$R+} ({$R–}), при этом устанавливается или отменяется генерация кода контроля вхождения в диапазон. В режиме {$R+} все индексы массивов и строк контролируются на выход за границы, а все присвоения значений скалярных переменных и ограниченных типов проверяются на вхождение в диапазон. При выходе за границы выполнение программы прекращается и выдается сообщение об ошибке.

Текст программы поиска номеров всех отрицательных элементов в массиве содержит строку: **For** i:=1 **To** n **Do**. Измените во второй строке n на $n+1$ и поставьте перед текстом програм-

мы директиву компилятора {$R+}. При запуске программы появится сообщение об ошибке периода выполнения: `Error 201: Range check error`. Она служит признаком того, что значение переменной *i* (индекс массива *A*) вышло за допустимый предел, то есть за значение константы *n*.

Примечание

При работе с массивами настоятельно рекомендуем использовать эту директиву при отладке программ.

3. Формирование значений элементов массива путем ввода их с клавиатуры — достаточно утомительное занятие. Давайте попробуем использовать для этих целей генератор случайных чисел. Это сложное название мы будем понимать очень просто. Есть что-то (черный ящик), и это что-то выдает числа, причем какое число выдается следующим за очередным, нам неизвестно. Этим черным ящиком в Паскале является функция Random. Она возвращает случайное число.

Функция Random.
Описание: Random[(*range*:Word)]. Напомним, что в квадратных скобках указывается необязательный параметр конструкции языка Паскаль.
Тип результата: Real или Word в зависимости от наличия параметра.
Комментарий. Если параметр не задан, то результатом является число типа Real в диапазоне $0 \le x < 1$. При наличии параметра возвращается число типа Word в диапазоне $0 \le x < range$. Обратите внимание на то, что верхняя граница диапазона не достигается, — неравенство строгое.

```
{$R-}
Program My10_4;
 Const n=10;
 Type MyArray=Array[1..n] Of Integer;
 Var A:MyArray;
     i:Integer;
 Begin
  {Randomize;}
  WriteLn('Формирование значений элементов
          массива A');
  For i:=1 To n Do A[i]:=Random(21){-10};
```

```
 WriteLn('Вывод');
 For i:=1 To n Do Write(A[i]:5);
 ReadLn;
End.
```

Выполним программу несколько раз. Каждый раз на экране мы видим одну и ту же последовательность чисел в диапазоне от 0 до 20. В строке текста программы, где записана функция Random, уберите фигурные скобки вокруг –10 и снова запустите программу несколько раз. Последовательность чисел на экране одна и та же, но теперь это числа из интервала от –10 до 10. Объясните этот факт. Попробуйте получать числа из любого интервала, например от –17 до 25.

Продолжим наши эксперименты. Уберите фигурные скобки у процедуры Randomize. Повторите многократный запуск программы. Последовательности чисел на экране разные. В чем дело? Наш черный ящик, функция Random, начинает генерировать в первом случае числа (каким-то неизвестным нам образом) от фиксированного начального числа. Во втором случае эти начальные числа меняются от запуска к запуску (процедурой Randomize) и последовательности получаются разные. Следующим изменением является включение контроля на выход за допустимый диапазон {$R+}. После запуска мы видим уже знакомую ошибку: `Error 201: Range check error`. Прочитайте еще раз внимательно описание функции Random и попробуйте найти объяснение причины возникновения ошибки. Если объяснение не найдено, то измените текст программы.

```
{$R+}
Program My10_4m;
 Const n=10;
 Var i:Integer;
 Begin
  WriteLn('Вывод 10 случайных чисел в диапазоне от
          -10 до 10');
  For i:=1 To n Do WriteLn(Random(21)-10);
  ReadLn;
 End.
```

Ошибки после запуска программы нет, но на экран выводятся не числа из интервала от –10 до 10, а какая-то странная последовательность. Она может быть, например, и такой: 65530, 1, 2, 4, 65527, 2, 65528, 65534, 3, 2. Где же истина? На-

помним, что диапазон для целых чисел типа Word простирается от 0 до 65 535, и число $65\ 535_{10}$ (то есть 2^{16}–1) равно 1111111111111111_2 , а это –1 в дополнительном коде. Число 65 534 соответствует –2 в дополнительном коде. В первой версии программы результат функции Random имеет тип Word. Из этого числа вычитается целое число типа Integer. Типы разные, их преобразование выполняется по правилу, описанному в занятии 2, то есть в данном случае все приводится к типу LongInt. А элемент массива имеет тип Integer — противоречие. Измените тип элементов массива *A* на LongInt и сравните результаты работы программ. Во второй версии контроль на выход за диапазон не выполняется — может быть, потому что нет операции присвоения (контролировать нечего и некого)? Верните тип Integer в описание элементов массива и измените одну строку программы на:

```
For i:=1 To n Do A[i]:=Integer(Random(21))-10;
```

Запустите программу. Она прекрасно работает. Объясните, что происходит в этом случае.

4. Научимся вычислять факториал натурального числа *n*. Факториал числа — это произведение чисел $1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$ (обозначается как *n*!). Сложность задачи в том, что уже 8!=40 320, 13!=6 227 020 800, а типы данных Integer и LongInt применимы в весьма ограниченном диапазоне натуральных чисел. Для представления факториала договоримся использовать массив.

Пример:

11!=39 916 800. Значения элементов массива *A* записаны в табл. 1.16.

Таблица 1.16

A[0]	*A*[1]	*A*[2]	*A*[3]	*A*[4]	*A*[5]	*A*[6]	*A*[7]	*A*[8]
8	0	0	8	6	1	9	9	3

Каким образом они представлены? В *A*[0] фиксируется число занятых элементов массива, в *A*[1] — цифра единиц результата, в *A*[2] — цифра десятков результата, в *A*[3] — цифра сотен результата и так далее. Почему так, а не наоборот? Такая запись позволяет исключить сдвиг элементов массива при переносе значений в старший разряд.

А сейчас наберите, как обычно, текст программы, выполните компиляцию и, открыв окно **Watch**, выполните ее в пошаговом режиме, отслеживая изменение значений переменных при не очень большом значении n. Добейтесь полного понимания логики работы программы.

```
{$R+}
Program My10_5;
 Uses Crt;
 Const MaxN=300;
 Type MyArray=Array[0..MaxN] Of Integer;
 Var A:MyArray;
    i,j,r,w,n:Integer;
 Begin
  ClrScr;
  FillChar(A,SizeOf(A),0); {Элементам массива A
                          присваивается значение 0.}
  WriteLn('Введите число, факториал которого
          необходимо подсчитать.');
  ReadLn(n);
  A[0]:=1;A[1]:=1;
  j:=2; {Начальные присвоения, начинаем вычислять
        2!.}
  While (j<=n) And (A[0]<MaxN) Do Begin {Второе
         условие фиксирует факт выхода за пределы
         массива.}
   r:=0;i:=1; {r - перенос из разряда в разряд при
        выполнении умножения числа j на очередную
        цифру A[i] предыдущего результата.}
   While (i<=A[0]) Or (r<>0) Do Begin {Пока не
        «прошли» все цифры предыдущего результата
        или есть перенос цифры в старший разряд.}
    w:=A[i]*j+r; {Умножаем очередную цифру
        и прибавляем цифру переноса из предыдущего
        разряда.}
    A[i]:=w Mod 10;{Находим новую цифру с номером i.}
    r:=w Div 10; {Вычисляем значение переноса.}
    If A[A[0]+1]<>0 Then A[0]:=A[0]+1; {Изменяем,
       если необходимо, количество задействованных
       элементов массива A, длину полученного
       результата.}
    i:=i+1;
```

```
    End;
    j:=j+1;
  End;
  For i:=A[0] DownTo 1 Do Write(A[i]);{Вывод
                                      результата.}
  WriteLn;
  ReadLn;
End.
```

Найдите число n, при котором 300-элементного массива A для хранения результата окажется недостаточно.

Измените программу так, чтобы выводились факториалы всех чисел в интервале от 1 до n.

Измените программу так, чтобы вычислялись степени чисел a^n, например 3^{100}.

Задания для самостоятельной работы

1. Дан массив целых чисел. Найдите:

- сумму элементов массива, больших данного числа a (a вводить с клавиатуры);
- сумму элементов массива, принадлежащих промежутку от a до b (a и b вводить с клавиатуры);
- максимальный элемент массива и его номер, при условии что все элементы различны;
- номера всех элементов массива с максимальным значением;
- все элементы массива с минимальным значением;
- сумму элементов массива, имеющих номера с $k1$-го по $k2$-й, где $k1$ и $k2$ вводятся с клавиатуры;
- количество нечетных элементов массива;
- количество отрицательных элементов массива;
- сумму первых пяти элементов массива;
- все элементы, кратные 3 или 5;
- сумму всех четных элементов массива, стоящих на четных местах, то есть имеющих четные номера;
- сумму всех четных элементов массива (или сумму элементов, кратных заданному числу);
- сумму положительных элементов массива;
- сумму элементов, имеющих нечетные значения;
- сумму элементов, имеющих нечетные индексы;

- сумму положительных элементов, значения которых меньше 10;
- удвоенную сумму положительных элементов;
- сумму отрицательных элементов;
- индексы тех элементов, значения которых больше заданного числа a;
- количество элементов массива, значения которых больше заданного числа a и кратны 5;
- индексы тех элементов, значения которых кратны 3 и 5;
- индексы тех элементов, значения которых больше значения предыдущего элемента (начиная со второго);
- количество тех элементов, значения которых положительны и не превосходят заданного числа a.

2. Определите:

- сколько элементов массива превосходят по модулю заданное число a;
- есть ли в данном массиве два соседних положительных элемента? Найти номера первой (последней) пары;
- есть ли в данном массиве элемент, равный заданному числу? Если есть, то вывести номер одного из них;
- есть ли в данном массиве положительные элементы, кратные k (k вводить с клавиатуры);
- номер первого отрицательного элемента, делящегося на 5 с остатком 2;
- пару соседних элементов с суммой, равной заданному числу;
- есть ли в массиве две пары соседних элементов с одинаковыми знаками;
- номер последней пары соседних элементов с разными знаками.

3. Измените программу вычисления факториала числа n так, чтобы вычислялись:

- $1 \cdot 3 \cdot 5 \cdot \ldots \cdot (2 \cdot n–1)$;
- $2 \cdot 4 \cdot 6 \cdot 8 \cdot \ldots \cdot (2 \cdot n)$.

4. Измените программу вычисления степени числа так, чтобы она могла вычислять степени отрицательных целых чисел.

5. Измените программу вычисления степени числа так, чтобы в степень возводились рациональные числа.

Занятие № 11. Процедуры

План занятия

1. Структура программы.
2. О взаимодействии между основной программой и процедурой.
3. Стандартные определения.
4. Эксперименты с программами (перестановка значений переменных a, b, c в порядке возрастания; вычисление выражения $y=a_n \cdot x^n+a_{n-1} \cdot x^n+...+a_2 \cdot x^2+a_1 \cdot x^1+a_0$; формирование значений элементов одномерного массива с помощью датчика случайных чисел; поиск элементов, принадлежащих двум массивам одновременно; перестановка частей массива).
5. Выполнение самостоятельной работы.

Структура программы

Программы на языке Паскаль состоят из заголовка программы, раздела описаний и тела программы. Раздел описаний может включать следующие подразделы: меток, констант, типов данных, переменных, процедур и функций. Последовательность подразделов в структуре программы произвольная, но естественно, что если вводится переменная нового типа, заданного в Type, то подраздел Type предшествует подразделу Var. Принцип нашего языка программирования «то, что используется, должно быть описано» сохраняется и для раздела описаний.

```
Program <имя программы>;
 Label <метки>;
 Const <описание констант>;
 Type <описание типов данных>;
 Var <описание переменных>;
 <процедуры и функции>;
 Begin
  <основное тело программы>;
 End.
```

До этого момента мы использовали в разделах описаний только описания переменных и типов. На этом занятии мы начнем изучать процедуры. Структура процедуры повторяет структуру программы. Отличие состоит в том, что у процедуры есть параметры.

```
Program имя процедуры (<параметры>);
 Label <метки>;
 Const <описание констант>;
 Type <описание типов данных>;
 Var <описание переменных>;
 <процедуры и функции>;
 Begin
  <основное тело процедуры>;
 End;
```

Примечание

В решениях задач на протяжении всей книги описание меток мы постараемся не использовать.

Какие вопросы возникают при знакомстве со структурами программы и процедуры? Их много. Начнем со взаимодействия программы и процедуры как по управлению, так и по данным.

Структура — взаиморасположение и связь составных частей чего-либо (из словаря иностранных слов). Итак, когда мы говорим о структуре, то обязаны сказать о том, из каких элементов она состоит и как они (элементы) связаны между собой. Следует также понимать, что структура обладает новыми свойствами, качествами по отношению к свойствам элементов, ее составляющих. Если структуре дать какое-то имя, то мы получаем нечто новое.

О взаимодействии между основной программой и процедурой

Для наглядности объяснения обратимся к примеру, приведенному на рис. 1.6. Слева приведен фрагмент текста основной программы, справа — процедура вычисления суммы двух целых чисел. Что мы видим? Процедура вызывается указанием

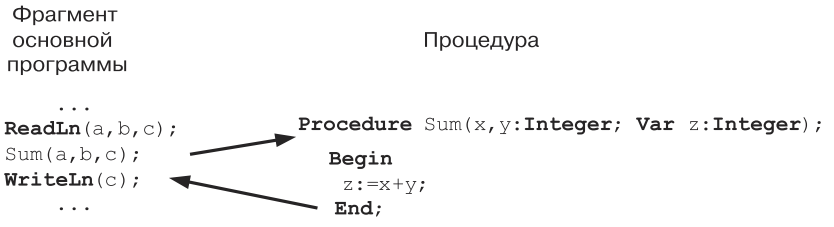


Рис. 1.6. Пример вызова процедуры

ее имени, которое записывается после зарезервированного слова Procedure, используемого для обозначения процедуры.

Линейный ход выполнения основной программы становится нелинейным — управление вычислительным процессом передается на участок программного кода, занимаемый процедурой. После выполнения процедуры происходит возврат на оператор основной программы, следующий за вызовом процедуры (End; в процедуре содержательный оператор — это «стрелка», возврат управления в логику, из которой вызывается процедура). Мы не рассматриваем вопросы о том, как это выполняется. Например, как происходит возврат на оператор WriteLn(*c*). Пока наше объяснение идет на уровне констатации фактов, и только.

Итак, взаимодействие программы и процедуры по управлению мы обозначили. Перейдем к взаимодействию по данным. В программе определены переменные *a*, *b*, *c*. В процедуре описаны ее параметры — *x*, *y*, *z*, но они являются переменными процедуры. Причем *x*, *y* описаны без идентификатора Var, а *z* — с идентификатором Var. Поясним разницу следующим рисунком (рис. 1.7). При вызове процедуры Sum(*a*,*b*,*c*) из основной про-

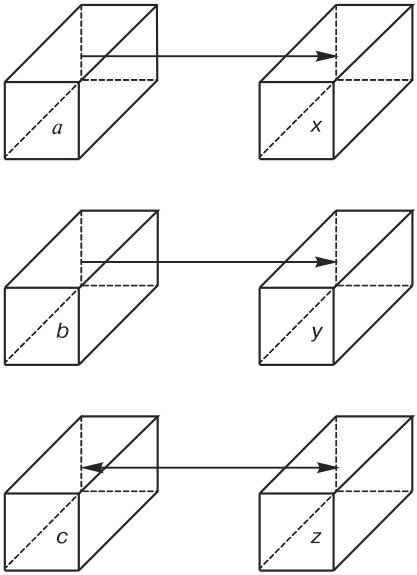


Рис. 1.7. Пример взаимодействия программы и процедуры по данным

граммы значение переменной *a* присваивается переменной *x* процедуры Sum, а значение переменной *b* становится значением переменной *y*, и все. Стрелка на рис. 1.7 в обоих случаях направлена в одну сторону. Чуть сложнее с переменными *c* и *z*: стрелка на рис. 1.7 направлена в обе стороны. Образно выражаясь, процедура Sum знает о том, где в памяти компьютера находится переменная *c*, и она знает, что при изменении значения переменной *z* необходимо произвести соответствующее изменение значения переменной *c*. Это обратная связь по данным от процедуры к основной программе. На такой тип связи и указывает идентификатор Var в описании параметров процедуры.

Стандартные определения

В любой программе все переменные делятся на глобальные и локальные. В нашем фрагменте программы переменные *a*, *b*, *c* — глобальные, а *x*, *y*, *z* — локальные. **Глобальные переменные** — это переменные из раздела описаний основной части программы, а **локальные переменные** — из раздела описаний процедур и функций. Если разница только в месте описания переменных, то «стоит ли шкурка выделки»? Для пояснения добавим следующее. Локальные переменные существуют только в течение времени работы процедуры — определяются (создаются) при ее вызове и исчезают после завершения работы процедуры.Таким образом, если бы в основной программе было, скажем, три фрагмента с вызовом процедуры Sum (желательно с различными параметрами), то для программного кода процедуры Sum (соответственно, и для переменных *x*, *y*, *z*) три раза выделялось бы место в оперативной памяти и три раза освобождалось. Если такое объяснение облегчает понимание разницы между глобальными и локальными переменными, то считаем его правильным.

Параметры. При описании процедуры указывается список **формальных** параметров. Каждый параметр является локальным по отношению к описываемой процедуре, к нему можно обращаться только в пределах данной процедуры (в нашем примере *x*, *y*, *z* — формальные параметры). **Фактические** параметры — это параметры, которые передаются процедуре при обращении к ней (*a*, *b*, *c* — фактические параметры). *Число и тип формальных и фактических параметров должны совпадать с точностью до их следования.*

Параметры-значения. Копия фактического параметра становится значением соответствующего формального параметра.

Внутри процедуры можно производить любые действия с данным формальным параметром (допустимые для его типа), но эти изменения никак не отражаются на значении фактического параметра, то есть каким он был до вызова процедуры, таким же и останется после завершения ее работы (x, y — параметры-значения).

Параметры-переменные. Это те формальные параметры, перед которыми стоит идентификатор Var. Передается адрес фактического параметра (обязательно переменной), после этого формальный параметр становится его синонимом. Любые операции с формальным параметром выполняются непосредственно над фактическим параметром.

Договоримся о том, как мы будем стараться использовать процедуры, может быть, в ущерб эффективности программ, например, с точки зрения количества переменных и так далее. Каждая процедура должна иметь одну точку входа и одну точку выхода, использование глобальных переменных в процедуре должно быть минимальным, взаимодействие вызывающей логики с процедурой должно осуществляться (по возможности) только через параметры. Почему? Мы осваиваем структурную технологию разработки программ, при этом на каждом этапе текущая задача разбивается на ряд подзадач, определяя тем самым некоторое количество отдельных подпрограмм (**подпрограмма** — это повторяющаяся группа операторов, оформленная в виде самостоятельной программной единицы). При этом мы стараемся структурировать задачу не только по управлению, но и по данным, используя при этом весьма ограниченный набор инструментов (параметры-значения, параметры-ссылки). Концепция процедур и функций — один из механизмов второго витка развития технологий программирования, а именно структурного проектирования.

Экспериментальный раздел занятия

1. Напишем программу перестановки значений переменных a, b, c в порядке возрастания, то есть так, чтобы $a<b<c$. Текст решения простой.

```
Program My11_1;
 Var a,b,c:Integer;
 Procedure Swap(Var x,y:Integer);
  Var t:Integer;
  Begin
```

```
    t:=x;
    x:=y;
    y:=t;
   End;
  Begin
   WriteLn('Введите три числа ');
   ReadLn(a,b,c);
   If a>b Then Swap(a,b);
   If b>c Then Swap(b,c);
   If a>c Then Swap(a,c);
   WriteLn(a:5,b:5,c:5);
   ReadLn;
  End.
```

Найдите ошибку в решении. Исправлению подлежит имя одной переменной, в одной строке программы. Составьте для поиска ошибки полную систему тестов. Измените программу так, чтобы аналогичная задача решалась для четырех вводимых чисел.

2. Составьте программу вычисления выражения

$$y=a_n \cdot x^n+a_{n-1} \cdot x^n+...+a_2 \cdot x^2+a_1 \cdot x^1+a_0,$$

где все данные — целые числа. Коэффициенты a_n, a_{n-1}, ..., a_1, a_0 являются первыми членами арифметической прогрессии, определяемой первым элементом q и разностью между соседними элементами d. Значения q, d, n, x вводятся с клавиатуры. Например, если q=2, d=3, n=5 и x=4, то нам необходимо вычислить выражение

$$2 \cdot 4^5+5 \cdot 4^4+8 \cdot 4^3+11 \cdot 4^2+14 \cdot 4^1+17 \cdot 4^0.$$

```
Program My11_2;
 Var  q,d,n,x,t,w,i: Integer;
      s: Integer;
 Procedure Degree(x,y: Integer; Var st: Integer);
  Var i:Integer;
  Begin
   st:=1;
   For i:=1 To y Do st:=st*x;
  End;
Begin
 WriteLn('Введите исходные данные - четыре
         не очень больших числа');
 ReadLn(q,d,n,x);
```

```
 s:=0;
 t:=n;
 For i:=1 To n+1 Do Begin
  Degree(x,t,w);
  s:=s+q*w;
  t:=t-1;
  q:=q+d;
 End;
 WriteLn('Результат ',s);
 ReadLn;
End.
```

Использование процедуры для решения этой задачи несколько искусственно, но пусть простит нас читатель. Наша цель — отработать механизм использования процедур. Последовательность действий здесь такова. Составьте таблицу изменения значений переменных q, t, w при работе программы. Проверьте правильность заполнения таблицы, используя отладчик и режим пошагового исполнения программы. Зафиксируйте значения q, d, x и найдите экспериментальным путем (или модифицируя программу) то значение n, при котором диапазона типа Integer не должно хватать для хранения результата.

Примечание

Переменные с именем x в основной программе и в процедуре Degree — это разные переменные!

Запишем приведенное выше выражение по-другому:

$17+4 \cdot (14+4 \cdot (11+4 \cdot (8+4 \cdot (5+4 \cdot 2))))$.

Результат вычислений, естественно, тот же самый. В общем виде запись выглядит следующим образом:

$a_0+x \cdot (a_1+x \cdot (a_2+x \cdot (...+x \cdot (a_{n-1}+x \cdot a_n))))$.

Приведенная запись называется схемой Горнера. Измените программу так, чтобы выражение вычислялось по схеме Горнера. Учтите, что в этом случае вам необходимо знать последний член арифметической прогрессии еще до первой итерации.

У нас есть две реализации одной и той же задачи. Сравните количество операций умножения и сложения, необходимых для получения результата в каждой из них. Какой вывод должны мы сделать?

3. Задание значений элементам одномерного массива (с помощью генератора случайных чисел) и вывод результата на экран мы рассмотрели на предыдущем занятии. Оформим эти действия как процедуры.

```
Program My11_3;
 Const n=8;l=-10;h=21;
 Type MyArray=Array[1..n] Of Integer;
 Var A:MyArray;
 Procedure Init(t,v,w:Integer;Var X:MyArray);
  Var i:Integer;
  Begin
   Randomize;
   For i:=1 To t Do X[i]:=v+Integer(Random(w));
  End;
 Procedure Print(t:Integer;X:MyArray);
  Var i:Integer;
  Begin
   For i:=1 To t Do Write(X[i]:5);
   WriteLn;
  End;
 Begin
  WriteLn('Формирование значений элементов
          массива A');
  Init(n,l,h,A); {Значения элементов массива A
              формируются из интервала целых чисел
              от l до h. Количество элементов n.}
  WriteLn('Вывод');
  Print(n,A); {n первых элементов массива A
             выводятся на экран (в данном случае).}
 End.
```

Казалось бы, что мы только проиграли. Было несколько строк программного кода, а сейчас... Но пожертвуем кажущейся простотой предыдущей версии программы. С этого момента будем стараться создавать программы так, чтобы основная программа состояла из вызовов процедур и функций.

Изменим заголовок процедуры Print на:

```
Procedure Print(n:Integer;X:Array[1..n] Of
                    Integer);
```

Запустим программу. Результат не заставит себя ждать. Это ошибка `Error 54: OF expected`. Курсор находится на

квадратной скобке, то есть после слова Array ожидается Of. «Пойдем на поводу» у системы программирования, изменим заголовок процедуры на:

```
Procedure Print(n:Integer;X:Array Of Integer);
```

Результат чуть-чуть изменился, уже знакомая ошибка — Error 201: Range check error. Отключим контроль на выход за пределы диапазона — {$R–}. Программа заработала, точнее, она выдает результат. Итак, сплошные загадки. Попробуйте дать им разумное объяснение.

4. Даны два одномерных массива. Найти элементы, принадлежащие одновременно обоим массивам.

Наша технология написания программ (использование процедур и функций) начинает приносить плоды. Процедуру Init при решении задачи требуется использовать два раза с различными параметрами, процедуру Print — три раза. Сделаем «костяк» программы. Процедуры Init и Print, естественно, не приводятся, они у нас почти универсальны и берутся из предыдущего задания.

Программа (ее «костяк») должна компилироваться. Сохраните ее. Набирать весь текст программы, а затем приступать к отладке — это «дурной» тон в программировании, очень «дурной». Возьмите за правило и всегда его придерживайтесь — «в любой момент времени у вас должна быть компилируемая программа, сохраненная на внешнем носителе». Кроме того, текст любой процедуры и, естественно, основной программы должен помещаться на экране и, конечно, быть читаемым.

```
{$R+}
Program My11_4;
 Const n=10;la=-10;ha=21;
       m=8; lb=-15;hb=31;
 Type MyArray=Array[1..n] Of Integer;
 Var A,B,C:MyArray;
     k:Integer;
 Procedure Solve(qx,qy:Integer;Var qz:Integer;
X,Y:MyArray;Var Z: MyArray);
  Begin
  End;
 Begin
```

```
  Init(n,la,ha,A);
  WriteLn('Вывод значений элементов массива A');
  Print(n,A);
  Init(m,lb,hb,B);
  WriteLn('Вывод значений элементов массива B');
  Print(m,B);
  Solve(n,m,k,A,B,C);
  WriteLn('Вывод тех целых чисел, которые есть и
          в A, и в B');
  Print(k,C);
  ReadLn;
 End.
```

Начнем уточнять наше решение, а именно процедуру Solve.

```
Procedure Solve(qx,qy:Integer;Var qz:Integer;
X,Y:MyArray;Var Z: MyArray);
  Var i,j:Integer;
  Begin
   qz:=0;
   For i:=1 To qx Do
    For j:=1 To qy Do
     If X[i]=Y[j] Then Begin
      qz:=qz+1;
      Z[qz]:=X[i];
     {j:=qy;}
    End;
  End;
```

После запуска программы окажется, что при некоторых исходных данных результат неправильный. Например, в первом массиве 10 присутствует один раз, во втором — пять раз. Согласно формулировке задачи, число 10 в ответе должно присутствовать один раз, а оно выводится пять раз. Уберем фигурные скобки у оператора `j:=qy` — «насильно» изменим управляющую переменную цикла `For j:=1`... Выводится правильный результат. Однако этот прием «насильственного», то есть не самим оператором For, изменения управляющей переменной считаем плохим программированием.

```
Procedure Solve(qx,qy:Integer;Var qz:Integer;
X,Y:MyArray;Var Z: MyArray);
```

```
Var i,j:Integer;
Begin
 qz:=0;
 For i:=1 To qx Do Begin
  j:=1;
  While (j<=qy) And (X[i]<>Y[j]) Do j:=j+1;
  If j<=qy Then Begin
   qz:=qz+1;
   Z[qz]:=X[i];
  End;
 End;
End;
```

Продолжите дальнейшее улучшение программы решения задачи.

5. Дан массив A из n элементов и число m ($1<m<n$). Не используя дополнительных массивов, переставить первые m элементов в конец массива, а элементы с $m+1$ по n — в начало, в остальном сохранив порядок элементов.

Идея решения: переворачиваем (записываем в обратном порядке) первые m элементов, затем переворачиваем элементы, начиная с $m+1$, и наконец переворачиваем все элементы получившегося массива.

Пример.

Пусть $n=10$, $m=6$ и массив A состоит из следующих чисел:

5 1 –4 3 7 2 –1 9 8 6

Далее массив преобразуется так:

2 7 3 –4 1 5 –1 9 8 6

– после первого переворачивания шести элементов.

2 7 3 –4 1 5 6 8 9 –1

– после второго переворачивания элементов с седьмого по десятый.

–1 9 8 6 5 1 –4 3 7 2

– после третьего переворачивания всех элементов массива.

В тексте решения не приводится реализация процедур Init и Print. Для процедуры Rev рекомендуется выполнить «ручную» трассировку при различных значениях t и l.

```
{$R+}
Program My11_5;
 Const n=10;m=6;
```

```
Type MyArray=Array[1..n] Of Integer;
Var A:MyArray;
Procedure Init(Var X:MyArray);
Procedure Print(X:MyArray);
Procedure Swap(Var a,b:Integer); {Из первого
                                  задания.}
Procedure Rev(t,l:Integer;Var X:MyArray);
Var i:Integer;
 Begin
  For i:=t To t+(l-t) Div 2 Do
   Swap(X[i],X[l-i+t]);
 End;
Begin
 Init(A);
 Print(A);
 Rev(1,m,A);
 Rev(m+1,n,A);
 Rev(1,n,A);
 Print(A);
 ReadLn;
End.
```

Задания для самостоятельной работы

1. Даны два различных выражения вида:

   $y=a_n \cdot x^n+a_{n-1} \cdot x^{n-1}+...+a_2 \cdot x^2+a_1 \cdot x^1+a_0$;
   $z=b_n \cdot x^n+b_{n-1} \cdot x^{n-1}+...+b_2 \cdot x^2+b_1 \cdot x^1+b_0$,

   где все данные — целые числа. Коэффициенты выражений хранятся в массивах A и B. Напишите программу нахождения максимального значения (y, z) при заданном значении x.

2. Даны два одномерных массива целых чисел. Найдите элементы, которые есть в первом массиве и которых нет во втором массиве.

3. Даны два одномерных массива целых чисел. Найдите элементы, которые входят только в один из массивов.

4. Из экспериментального раздела занятия решить задачу 4 для трех массивов.

5. Даны три одномерных массива целых чисел. Найдите элементы, которые есть в первом массиве и которых нет во втором и третьем массивах.
6. Решите задачу 3 из экспериментального раздела для трех одномерных массивов.
7. Даны два одномерных массива целых чисел с разным количеством элементов. Найдите среднее арифметическое элементов каждого массива и их сумму. Решите задачу для трех массивов.
8. Даны три одномерных массива целых чисел одинаковой размерности. Сформируйте четвертый массив, каждый элемент которого равен максимальному из соответствующих элементов первых трех массивов.
9. Даны два одномерных массива (A, B) одинаковой размерности n и число m. Поменяйте местами первые элементы массивов и выполните перестановку элементов массивов в обратном порядке по образцу задачи 5 из экспериментального раздела.
10. Дано четное число n из интервала $2<n\leq 999$. Проверьте для этого числа гипотезу Кристиана Гольдбаха (1742 г.). Эта гипотеза (по сегодняшний день не опровергнутая и полностью не доказанная) заключается в том, что каждое четное n, большее двух, представимо в виде суммы двух простых чисел. Оформите в виде процедуры проверку, является ли число простым.

    Примеры:

    $6=3+3$, $12=5+7$, $30=7+23$, $308=31+277$, $992=73+919$.

    Проверьте гипотезу Гольдбаха для бóльших значений n.

11. Известны следующие признаки делимости чисел:
    - для делимости на 2 необходимо, чтобы последняя цифра числа делилась на 2;
    - для делимости на 3 требуется, чтобы сумма цифр числа делилась на 3;
    - для делимости на 4 необходимо, чтобы число из последних двух цифр делилось на 4;
    - для делимости на 5 необходимо, чтобы последняя цифра числа была равна 0 или 5;
    - для делимости на 8 необходимо, чтобы число из 4 последних цифр делилось на 8;

- для делимости на 9 необходимо, чтобы сумма цифр числа делилась на 9;
- для делимости на 11 необходимо, чтобы разность между суммой цифр, стоящих на четных местах, и суммой цифр, стоящих на нечетных местах, делилась на 11.

Напишите процедуры проверки признаков делимости для числа n. Проверьте их для различных значений n.

Занятие № 12. Функции

План занятия

1. Функции.
2. Сведения из комбинаторики.
3. Эксперименты с программами (вычисление числа сочетаний из n по m; получение палиндрома из числа путем последовательного его «перевертывания» и сложения; нахождение общей части n отрезков).
4. Выполнение самостоятельной работы.

Функции

Функция так же, как и процедура, используется для структурирования программ. В некоторых языках вообще нет разделения подпрограмм на процедуры и функции. В Паскале такое разделение есть, но носит преимущественно синтаксический характер.

Заголовок функции начинается с ключевого слова Function и кончается типом возвращаемого данной функцией значения.

```
Function <имя функции> [(список параметров)]:<тип результата>;
```

В теле функции обязательно должен быть хотя бы один оператор присвоения, где в левой части стоит имя функции, а в правой — ее значение. Иначе значение функции не будет определено.

Сведения из комбинаторики

Перестановки. Перестановкой из n элементов называется упорядоченный любым возможным способом набор из n раз-

личных натуральных чисел, принадлежащих интервалу от 1 до n.

Пусть n=3. Перечислим все перестановки из 3 элементов: (1 2 3), (1 3 2), (2 1 3), (2 3 1), (3 1 2), (3 2 1).

Количество перестановок из n элементов (обозначается как P_n) равно $n!=1 \cdot 2 \cdot \ldots \cdot n$ (факториалу числа n). Действительно, есть n способов для выбора элемента на первое место, $(n-1)$ способ для выбора элемента на второе место и так далее. Общее количество способов — это произведение $n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1$.

Перечислите все перестановки из 4 элементов. Количество перестановок в данном случае — 24. Подсчитайте вручную значения P_n для n из диапазона от 5 до 12.

Размещения. Размещением из n элементов по m называется любая возможная выборка m чисел, принадлежащих интервалу от 1 до n (как обычно, рассматриваем натуральные числа). Два размещения считаются различными, если они либо отличаются друг от друга хотя бы одним элементом, либо состоят из одних и тех же элементов, но расположенных в разном порядке.

Размещением с повторениями называется выборка, в которую могут входить и одинаковые элементы (числа).

Пусть n=3, а m=2. Перечислим все размещения с повторениями из 3 по 2: (1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3).

В общем случае число таких размещений вычисляется по формуле $R(n, m)=n^m$. Это доказывается с помощью математической индукции.

Перечислите все размещения с повторениями при n=4 и m=2. Подсчитайте вручную значения $R(n, m)$ для нескольких значений n и m.

Размещением без повторений из n элементов по m называется выборка из m *различных* чисел, принадлежащих интервалу от 1 до n.

Пусть n=4, а m=2. Перечислим все размещения без повторений из 4 по 2: (1, 2), (2, 1), (1, 3), (3, 1), (1, 4), (4, 1), (2, 3), (3, 2), (2, 4), (4, 2), (3, 4), (4, 3).

В общем случае количество таких размещений вычисляется по формуле $A(n, m)=n \cdot (n-1) \cdot \ldots \cdot (n-m+1)$ или $A(n,m) = \dfrac{n!}{(n-m)!}$.

Перечислите все размещения без повторений из 5 элементов по 2. Подсчитайте вручную значения $A(n, m)$ для фиксиро-

ванного значения m, например 4, и n, принимающих значения от 8 до 13.

Сочетания. Сочетанием из n элементов по m называется множество из m *различных* чисел, выбранных из диапазона от 1 до n. Два множества считаются различными, только если они отличаются друг от друга хотя бы одним элементом. Наборы из одних и тех же элементов, хотя бы и расположенных в разном порядке, — это одно и то же множество (сочетание).

Пусть n=5, а m=3. Перечислим все сочетания из 5 по 3: (1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5), (1, 4, 5), (2, 3, 4), (2, 3, 5), (2, 4, 5), (3, 4, 5).

Каждому сочетанию $C(n, m)$ соответствует $m!$ размещений (за счет перестановок элементов). Таким образом, $A(n, m) = C(n, m) \cdot m!$, откуда $C(n,m) = \dfrac{A(n,m)}{m!} = \dfrac{n!}{m!\cdot(n-m)!}$.

Перечислите все сочетания из 7 элементов по 5. Подсчитайте число сочетаний для различных значений n и m (не очень больших). При подсчете как числа сочетаний, так и числа размещений используется операция деления. Обоснуйте, почему в результате получаются только целые числа.

Экспериментальный раздел занятия

1. Составим программу подсчета числа сочетаний $C(n, m)$. Пусть у нас есть функция подсчета факториала числа — Fact(n). Написание основной программы сводится к программированию формулы:

$$C(n,m) = \frac{n!}{m!\cdot(n-m)!}.$$

Обратите внимание на то, что вызывать функцию можно прямо при вычислении выражения, а не в отдельной строке программы, как мы делали при вызове процедур.

```
Program My12_1;
 Var  n,m: Integer;
      c: Longint;
 Function Fact(n:Integer):Longint;
  Begin
  End;
Begin
 WriteLn('Введите n и m :');
 ReadLn(n,m);
```

```
 c:=Fact(n) / (Fact(m) * Fact(n-m));
 WriteLn(c);
 ReadLn;
End.
```

При компиляции программы (функцию Fact мы пока не писали, но компилироваться программа должна и без нее) в строке

```
c:=Fact(n) / (Fact(m) * Fact(n-m));
```

возникает ошибка: `Error 26: Type mismatch` — несоответствие типов. Вспомним, что типы данных определяют не только диапазон значений переменных, но и допустимые над этим типом операции. Обычное деление «/» недопустимо для переменных целого типа. Необходимо использовать операцию Div:

```
c:=Fact(n) Div (Fact(m) * Fact(n-m));
```

Дополняем программу.

```
Function Fact(n:Integer):Longint;
 Var i: Integer;
    rez: Longint;
 Begin
  rez:=1;
  For i:=1 To n Do rez:=rez*i;
  Fact:=rez;
 End;
```

Запускаем ее для различных значений m и n.

При n=10, m=5 результат 252 пока не вызывает сомнений.

При n=13, m=5 результат 399 — что-то не так, должен быть больше.

При n=15, m=7 результат 9 — программа явно не работает. Объясните причину. Подсказка содержится в материале этого занятия.

А можно ли упростить программу? Переменная c — лишняя. Попробуйте убрать ее.

Очевидно также, сколько лишних вычислений (умножений) делает эта программа. При подсчете

$$C(13,5) = (1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot 8 \cdot 9 \cdot 10 \cdot 11 \cdot 12 \cdot 13)/ \\ ((1 \cdot 2 \cdot 3 \cdot 4 \cdot 5) \cdot (1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 \cdot 7 \cdot 8))$$

разумный человек вряд ли будет выполнять все умножения. Он выполнит $9 \cdot 11 \cdot 13$ и сразу получит 1287. Исходя из этого, напи-

шите более эффективную функцию вычисления числа сочетаний. Например, даже нижеприведенный вариант программы даст более обнадеживающие результаты. Правильный результат при $C(13,5)$, а именно 1287 — уже шаг вперед, но это не предел улучшений программы.

```
Program My12_1m;
 Var n,m:Integer;
 Function S(n,m:Integer):LongInt;
  Var i:Integer;
      rez,cht:LongInt;
  Begin
   rez:=1;
   cht:=1;
   For i:=1 To m Do Begin
     rez:=rez*i;
     cht:=cht*(n-i+1);
    End;
   S:=cht Div rez;
  End;
 Begin
   WriteLn('Введите два числа: ');
   ReadLn(n,m);
   WriteLn(S(n,m));
   ReadLn;
 End.
```

Продолжим тему функций, хотя при решении модифицированной задачи они и не используются.

Познакомимся с **основной теоремой арифметики**. *Всякое натуральное число n>1 разлагается на произведение простых чисел. Это разложение однозначно с точностью до порядка записи простых чисел в произведении.*

Пусть нам не требуется вычислять $C(n,m)$ и необходимо представить это число в виде $p_1^{\alpha 1} p_2^{\alpha 2} \ldots p_q^{\alpha q}$, где $\alpha_i \geq 0$, а p_i — простые числа. Если даже значение n не превышает 100, то вычислять, что называется, «в лоб» — безнадежное занятие. Числа огромные. Рассмотрим идею решения на примере вычисления $C(8,5)=8!/(5! \cdot 3!)$.

Представим все сомножители 8! как элементы в соответствующем массиве, например с именем Sn. Запись $Sn[i]=1$ говорит о том, что сомножитель i есть в итоговом произведении.

Просматриваем массив *Sn*, выбираем очередной элемент — число *i*. Если оно составное, то находим его разложение на простые сомножители. К элементам *Sn*, соответствующим этим простым сомножителям, прибавляем количество единиц, равное числу их вхождений в *i*, а *Sn*[*i*] обнуляем. В итоге получаем разложение числа 8! только на простые множители.

Аналогично представим все сомножители 5! как элементы массива *Sm*, сомножители 3! — как элементы массива *Snm* и выполним с ними те же действия (см. табл. 1.17). После этого останется только провести вычитание степеней одного и того же простого числа в затененных строках таблицы.

Таблица 1.17

	i	2	3	4	5	6	7	8
8!	***Sn***	1	1	1	1	1	1	1
	составной сомножитель 4	3	1	0	1	1	1	1
	составной сомножитель 6	4	2	0	1	0	1	1
	составной сомножитель 8	7	2	0	1	0	1	0
5!	***Sm***	1	1	1	1	0	0	0
		3	1	0	1	0	0	0
3!	***Snm***	1	1	0	0	0	0	0
C(8,5)	После вычитания	3	0	0	0	0	1	0

Итак, $C(8,5)=2^3*7^1$.

```
Program My12_2;
 Const Q=100;
 Type MyArray=Array[1..Q] Of Integer;
 Var n,m:Integer;
     P:MyArray;
 Procedure Solve(sn,sm:Integer;Var Sp:MyArray);
  Begin
  End;
 Procedure Print(pn:Integer;Pp:MyArray);
  Var i:Integer;
  Begin
   For i:=2 To pn Do
    If Pp[i]<>0 Then WriteLn(i:5,Pp[i]:5);
  End;
 Begin
  WriteLn('Ввод чисел n и m');
  ReadLn(n,m);
```

```
 Solve(n,m,P);
 Print(n,P);
 ReadLn;
End.
```

Пока у нас есть только основная программа, содержащая вызовы процедур, и простая процедура Print. Программа компилируется. Программируем так, как говорим: ввели данные; выполнили обработку; вывели результат. Придерживайтесь этого принципа написания. Язык программирования для вас должен стать естественным языком для выражения мыслей, так же как, например, русский язык в общении. Уточнение логики.

```
Procedure Solve(sn,sm:Integer;Var Sp:MyArray);
 Var i:Integer;
    Sni,Smi,Snmi:MyArray;
Begin
 Calc(sn,Sni); {Сформировали массив,
                соответствующий n!}
 Calc(sm,Smi); {Сформировали массив,
                соответствующий m!}
 Calc(sn-sm,Snmi); {Сформировали массив,
                    соответствующий (n-m)!}
 For i:=2 To sn Do Sp[i]:=Sni[i]-Smi[i]-Snmi[i];
End;
```

Сделав заголовок процедуры Calc с операторами Begin и End, вы получите опять компилируемую программу. Вариант процедуры Calc, приведенный ниже, хотя и работает, но не так хорошо читабелен, как предыдущая логика. Выполните трассировку этой процедуры в режиме отладки, определите назначение каждого оператора. Ваша задача — улучшить процедуру. Вспомните о том, что в материале занятия 8 перечислены все простые числа до 100.

```
Procedure Calc(n:Integer; Var X:MyArray);
 Var i,j,t:Integer;
 Begin
  FillChar(X,SizeOf(X),0); {SizeOf - вычисляет
  размер области памяти, выделенной для массива X;
  FillChar - записывает 0 в эту область памяти
  (инициализация X).}
```

```
 For i:=1 To n Do X[i]:=1; {Признаки чисел,
  задействованных в вычислении факториала числа}
 For i:=4 To n Do Begin
  t:=i;
  j:=2;
  While (j<=(t Div 2)) Do
   If t Mod j=0 Then Begin {j делит t без остатка}
     X[j]:=X[j]+1;
     t:=t Div j;
     If t=j Then X[j]:=X[j]+1;
    End
   Else j:=j+1;
  If t<>i Then Begin {Найдены делители числа i}
   X[i]:=0;
   If (t<>j) Then X[t]:=X[t]+1;
  End;
 End;
End;
```

Попробуем теперь исследовать задачу для большего диапазона значений n. Пусть $n \le 10000$. Запуск программы принесет только разочарование, даже если изменить тип Integer на LongInt. Появится ошибка `Error 202: Stack overflow error`. Не занимаясь детальным выяснением ее сути, считаем, что просто не хватает оперативной памяти компьютера для хранения данных программы. Действительно, массивов введено много. Достаточно одного массива P при условии, что он объявляется как глобальный и не используется при вызове процедур. Вариант программы:

```
{$R+}
Program My12_2m;
 Const Q=10000;
 Type MyArray=Array[1..Q] Of Integer;
 Var n,m:LongInt;
     P:MyArray;
 Procedure Inc_O(n:LongInt);
  Var i,j,t:Integer;
  Begin
   For i:=2 To n Do Begin
    t:=i;
```

```
    j:=2;
    While (j<=(t Div 2)) Do
     If t Mod j=0 Then Begin
        P[j]:=P[j]+1;
        t:=t Div j;
        If t=j Then P[j]:=P[j]+1;
       End
      Else j:=j+1;
     If t<>i Then Begin
        If (t<>j) Then P[t]:=P[t]+1;
       End
      Else P[i]:=1;
   End;
  End;
 Procedure Dec_O(n:LongInt);
  Var i,j,t:Integer;
  Begin
   For i:=2 To n Do Begin
    t:=i;
    j:=2;
    While (j<=(t Div 2)) Do
    If t Mod j=0 Then Begin
       P[j]:=P[j]-1;
       t:=t Div j;
       If t=j Then P[j]:=P[j]-1;
      End
     Else j:=j+1;
    If t<>i Then Begin
      If (t<>j) Then P[t]:=P[t]-1;
     End
    Else P[i]:=P[i]-1;
   End;
  End;
 Begin
  WriteLn('Ввод чисел n и m'); ReadLn(n,m);
  Inc_O(n);
  Dec_O(m);
  Dec_O(n-m);
 End.
```

Программа работает, но она опять не лучшая (вывод результата не приводится). Код процедур Inc_O и Dec_O почти повторяется. Это плохо. Избавьтесь от этого недостатка, а может быть, вами уже сделан более достойный вариант обработки. В этой программе значение константы *Q* можно увеличить, например, до 30 000. Возникает другая проблема — время работы программы: приходится долго ждать результат.

2. Задание № 11 занятия № 7 требовало определить, является ли введенное число палиндромом.

Оформим «перевертывание» числа в виде функции.

```
Function  Pal(n:LongInt):LongInt;
 Var x:LongInt;
 Begin
  x:=0;
  While n<>0 Do Begin
    x:=x*10+n Mod 10;
    n:=n Div 10;
   End;
  Pal:=x;
 End;
```

Пусть введено исходное число, например 59. Оно не палиндром. «Перевернем» его, получим 95. Найдем сумму чисел 59 и 95 — 154. «Перевернем» это число — 451. Находим сумму — 605. Еще раз: 506 и 1111. Получили палиндром. Найти для всех натуральных чисел из интервала от 50 до 80 количество шагов, необходимых для сведения их к палиндромам с помощью описанной схемы.

```
Program My12_3;
 Const a=50;b=80;
 Var n,t:LongInt;
     cnt,i:Integer;
Function  Pal(n:LongInt):LongInt;
  Begin
  …
  End;
 Begin
  For i:=a To b Do Begin
    cnt:=0;
    n:=i;
```

```
    While Pal(n)<>n Do Begin
      t:=Pal(n);
      n:=n+t;
      Write(t,' ',n,' ');
      cnt:=cnt+1;
    End;
    WriteLn(i,' ',cnt);
   End;
End.
```

Попробуйте получить палиндром из числа 89 (или числа 98). На 16-м шаге результат по-прежнему не является палиндромом, а сумма выходит за пределы типа LongInt. Измените программу, используя пример 4 из экспериментального раздела занятия № 10, и исследуйте проблему. Имеет ли она решение?

3. Из занятия № 8 мы узнали, как находить наибольший общий делитель двух чисел. Оформим эти действия в виде функции.

```
Function Nod(a,b:LongInt):LongInt;
 Begin
  Repeat
   If a>b Then a:=a Mod b Else b:=b Mod a;
  Until (a=0) Or (b=0);
  Nod:=a+b;
 End;
```

Рассмотрим следующую задачу. Дано n отрезков, длины которых — целые числа. Берем два отрезка и из большего вычитаем меньший. Разность — новый отрезок. Если же длины отрезков совпадают, то один из них исключаем из дальнейшего рассмотрения. Продолжаем процесс. Ответом задачи является одно число, равное длине оставшегося в результате этих действий отрезка.

Пример:

Число отрезков n равно 4, их длины равны 6, 15, 3 и 9 (табл. 1.18).

Таблица 1.18

	6	**15**	**3**	**9**
1-й шаг	6	9	3	9
2-й шаг	6	3	3	9

3-й шаг	3	3	3	9
4-й шаг	–	3	3	9
5-й шаг	–	–	3	9
6-й шаг	–	–	3	6
7-й шаг	–	–	3	3
8-й шаг	–	–	–	3

Решение задачи «в лоб» по приведенной в табл. 1.18 схеме возможно, хотя оно достаточно трудоемко и для больших значений n требует больших временных затрат. На самом деле вся описанная процедура вычитаний сводится к нахождению наибольшего общего делителя n чисел, так что сами вычитания выполнять не требуется. Напишите программу решения задачи. Исследуйте ее работоспособность для больших значений n.

Задания для самостоятельной работы

1. Дано n целых чисел. Найдити среди них числа, у которых приведенная ниже характеристика имеет максимальное значение:
   - сумма цифр;
   - первая цифра;
   - количество делителей;
   - сумма всех делителей.
2. Дано n целых чисел. Найдите среди них пару чисел, для которых выполняется приведенное ниже условие:
   - наибольший общий делитель имеет максимальное значение;
   - наименьшее общее кратное имеет наименьшее значение;
   - есть ли среди заданных чисел «близнецы», то есть простые числа, разность между которыми равна двум.
3. Дано n целых чисел. Напишите программу подсчета количества чисел, в записи которых нет цифры 8. При этом требуется использовать функцию вида:

```
Function Yes8(x:LongInt):Boolean;
```

Без использования компьютера решите эту задачу для всех трехзначных чисел. Ответ: 729.

4. Даны два числа n и k ($k<n$). При этом n означает основание системы счисления, а k — количество знаков в записи числа. Напишите программу подсчета количества натураль-

ных чисел в системе счисления с основанием n, записываемых ровно k знаками. Без использования компьютера решите эту задачу для всех трехзначных чисел в десятичной системе счисления. Ответ: 900.

5. Объявлен конкурс на получение грантов. В конкурсе принимают участие n человек. Гранты выдаются первым пяти участникам, набравшим наибольшее количество баллов. Один участник не может одновременно получить два и более грантов. Напишите программу определения количества способов, которыми могут быть распределены гранты. Исследовать ее область применимости.

6. Дана шахматная доска размером $n \times n$ и n ладей. Определите количество способов расположения всех n ладей на доске так, чтобы они не могли бить друг друга. Исследуйте, для каких значений n работоспособны типы данных Integer, Word, LongInt.

7. Дано n бусин. Сколько различных ожерелий можно составить из них? Исследуйте область применимости вашей программы.

Примечание

При n=7 число ожерелий равно (7!:7):2=360. Делением на 7 учитывается поворот «по кругу», делением на 2 — переворот ожерелий, то есть операции, которые не создают нового ожерелья.

8. Имеются предметы k различных типов. Количество предметов первого типа — n_1, второго — n_2, ..., k-го — n_k. Напишите программу подсчета числа перестановок, которые можно сделать из этих предметов, и исследовать ее область применимости. Предварительно попытайтесь обосновать формулу:

$$P_n = \frac{n!}{n_1! \cdot n_2! \cdot \ldots \cdot n_k!}, \text{ где } n = n_1 + n_2 + \ldots + n_k.$$

Есть текст. Его шифруют с помощью перестановки букв. В результате получается новый текст, называемый анаграммой (слова лунка и кулан — анаграммы). Для механической расшифровки анаграммы требуется выполнить количество перестановок, вычисляемое по вышеприведенной формуле, где n_i — количество конкретных букв в анаграмме. Предположим, что у вас есть компьютер с миллиардным быстродействием (1 миллиард операций, в частности миллиард умножений в секунду). Задайте значения вели-

чин n, n_1, n_2, ..., n_k и подсчитайте, за какое время ваш компьютер решит задачу.

9. Черепашка перемещается по клеточному полю размером $n \times m$. Ей разрешено двигаться вправо и вверх. В начальный момент времени она находится в нижней левой клетке, и ей необходимо попасть в верхнюю правую клетку. Сколько различных путей есть у черепашки? Подсчитайте количество путей для различных значений n и m. Выясните, для каких значений n и m работоспособны типы данных Integer, Word, LongInt, то есть для хранения результата можно использовать переменные этих типов. Предположим, что для анализа одного пути черепашки компьютеру с миллиардным быстродействием требуется 100 операций. За какое время он проанализирует все пути для конкретных значений n и m?

10. Двоичные последовательности длины n составляются из t нулей и k единиц, причем таким образом, что никакие две единицы не находятся рядом. Подсчитайте количество таких последовательностей (ответ — C_{t+1}^{k}). Найдите способ перечисления всех таких последовательностей для небольших значений t и k.

Пример:

t=3, k=2. Последовательности: 00101, 01001, 01010, 10001,10010,10100.

Занятие № 13. Рекурсия

План занятия

1. Понятие рекурсии.

2. Короткие примеры для иллюстрации рекурсии.

3. Эксперименты с программами (перечисление всех последовательностей длины n из чисел от 1 до k; генерация перестановок чисел от 1 до n; перечисление всех разбиений числа n на сумму слагаемых $n=a_1+a_2+ \ldots +a_k$, где k, a_1, a_2, ..., $a_k>0$).

4. Выполнение самостоятельной работы.

Понятие рекурсии

Рекурсией называется ситуация, когда процедура или функция вызывает сама себя. Типичная конструкция рекурсивной процедуры имеет вид:

```
Procedure Rec(t:Integer);
 Begin
  <действия на входе в рекурсию>;
  If <проверка условия> Then Rec(t+1);
  <действия на выходе из рекурсии>;
 End;
```

При написании рекурсивной логики необходимо осознать два ключевых момента: ход вычислительного процесса по управлению; связи в вычислительном процессе по данным. Проиллюстрируем указанные моменты для конкретной задачи вычисления факториала числа (рис. 1.8).

```
Function Factorial( n: Integer): Longint;
 Begin
  If n=1 Then Factorial:=1
   Else Factorial:=n* Factorial(n-1);
 End;
```

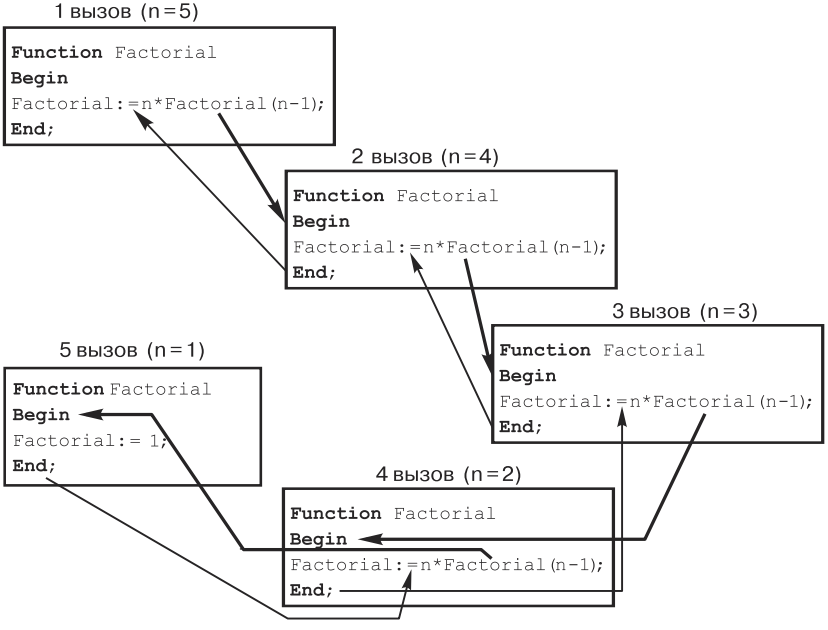


Рис. 1.8. Схема рекурсивных вызовов при вычислении 5!

Найдем 5!. Как вычисляется факториал числа? Первый вызов функции производится из основной программы, например a:=Factorial(5) — переменной *a* присваиваем значение 5!. Действий на входе в рекурсию нет, этап вхождения обозначен на рисунке жирными линиями. Он продолжается до тех пор, пока значение локальной переменной не становится равным 1. После этого начинается выход из рекурсии (тонкие линии на рис. 1.8). Итак, ход вычислительного процесса по управлению является нелинейным. В рекурсивной логике обязательно должно быть условие завершения процесса вхождения в рекурсию. Естественный вопрос — что необходимо знать для реализации этого процесса? С входом в рекурсию более понятно — производится вызов процедуры или функции, а для реализации выхода из рекурсии требуется помнить, откуда мы пришли, то есть помнить точки возврата в вызывающую логику. Чтобы помнить, необходимо хранить. Это место, место хранения точек возврата, называется «стеком вызовов», и для него нашей системой программирования выделяется определенная область оперативной памяти. В этом стеке запоминаются не только адреса точек возврата, но и значения всех локальных переменных, — образно выражаясь, запоминается «слепок» процедуры или функции. По «слепку» восстанавливается вызывающая логика (процедура или функция). Как только мы осознаем это утверждение, становятся понятными связи по данным в рекурсивном вычислительном процессе.

Примечание

Написание рекурсивных процедур по образцу, приведенному ниже, вряд ли можно считать приемлемым.

```
Procedure Generate;
 Begin {t - глобальная переменная}
  If t=n Then Exit
   Else Begin
      <действия 1>;
      t:=t+1;
      Generate;
      t:=t-1;
      <действия 2>;
     End;
 End;
```

Безусловно, при данной реализации экономится место в стеке вызовов (стеке адресов возврата), но это простая замена цикла на рекурсию.

Короткие примеры для иллюстрации рекурсии

1. Сложение двух чисел ($a+b$). Рекурсивное определение операции сложения двух чисел:

$$a+b=\begin{cases} a & \text{при } b=0; \\ (a+1)+(b-1) & \text{при } b>0; \\ (a-1)+(b+1) & \text{при } b<0. \end{cases}$$

```
Function Sum( a, b: Integer): Integer;
 Begin
  If b=0 Then Sum:=a
   Else
    If b>0 Then Sum:=Sum( a+1, b-1)
     Else Sum:=Sum( a-1, b+1 );
 End;
```

2. Перевод натурального числа из десятичной системы счисления в двоичную.

```
Procedure Rec( n: Integer);
 Begin
  If n>1 Then Rec(n Div 2);
  Write(n Mod 2);
 End;
```

Как изменится процедура при переводе чисел в восьмеричную систему счисления?

3. Определение — является ли заданное натуральное число простым. Задачу можно сформулировать по-другому: верно ли, что заданное натуральное число n не делится ни на одно число, большее или равное m, но меньшее n, если значения числа m находятся в промежутке от 2 до n? Утверждение истинно в двух случаях:
   - если $m=n$;
   - если n не делится на m, и это утверждение истинно для чисел от $m+1$ до $n-1$.

```
Function Simple(m,n: Integer): Boolean;
 Begin
  If m=n Then Simple:=True
   Else Simple:=(n Mod m<>0) And Simple(m+1,n);
 End;
```

Первый вызов функции: Simple(2, n), где n – проверяемое число.

4. Дано натуральное число $n \geq 1$. Определить число a, для которого выполняется неравенство: $2^{a-1} \leq n < 2^a$. Зависимость:

$$a(n) = \begin{cases} 1 & \text{при } n = 1; \\ a \cdot (n \text{ Div } 2) + 1 & \text{при } n > 1. \end{cases}$$

```
Function a( n: Integer): Integer;
 Begin
  If n=1 Then a:=1 Else a:=a(n Div 2) +1;
 End;
```

5. Для вычисления наибольшего общего делителя двух чисел можно использовать рекурсивную функцию вида:

```
Function Nod(a,b:Integer):Integer;
 Begin
  If (a=0) Or (b=0) Then Nod:=a+b
   Else
    If a>b Then Nod:=Nod(a-b,b)
     Else Nod:=Nod(a,b-a);
 End;
```

6. Нахождение максимального элемента в глобальном массиве A реализуется с помощью следующей рекурсивной логики.

```
Procedure Search_Max(n:Integer; Var x:Integer);
 Begin
  If n=1 Then x:=A[1]
   Else Begin
     Search_Max(n-1,x);
     If A[n]>x Then x:=A[n];
   End;
 End;
```

7. Возведение целого числа a в целую неотрицательную степень n рекурсивно реализуется так:

```
Function Pow(a,n:Integer):Integer;
 Begin
  If n=0 Then Pow:=1
   Else
    If n Mod 2=0 Then Pow:=Pow(a*a,n Div 2)
     Else Pow:=Pow(a,n-1)*a;
 End;
```

8. Процедура ввода с клавиатуры последовательности чисел (окончание ввода — 0) и вывода ее на экран в обратном порядке имеет вид:

```
Procedure Solve;
 Var n:Integer;
 Begin
  ReadLn(n);
  If n<>0 Then Solve;
  Write(n:5);
 End;
```

9. Каждое число в ряду Фибоначчи равно сумме двух предыдущих чисел ряда при условии, что первые два равны 1 (1, 1, 2, 3, 5, 8, 13, 21,...). В общем виде n-е число Фибоначчи можно определить так:

$$f(n)=\begin{cases}1, & \text{если } n=1 \quad \text{или} \quad n=2;\\ f(n-1)+f(n-2), & \text{если } n>2.\end{cases}$$

```
Function Fib(n:Integer):Integer;
 Begin
  If n<=2 Then Fib:=1
   Else Fib:=Fib(n-1)+Fib(n-2);
 End;
```

10. Нахождение суммы первых n членов арифметической (геометрической) прогрессии. Арифметическая прогрессия определяется тремя параметрами: первым элементом (a), разностью между соседними элементами (d) и коли-

чеством членов прогрессии (n). Очередной элемент прогрессии вычисляется из предыдущего прибавлением разности:

$a_{\text{новое}} := a_{\text{старое}} + d$.

```
Function Sa(n,a:Integer):Integer;
  Begin
   If n>0 Then Sa:=a+Sa(n-1,a+d)
    Else Sa:=0;
  End;
```

Примечание

Попробуйте убрать ветвь Else и объяснить полученный результат.

Для приведенных примеров прорисуйте схемы работы процедур и функций для конкретных входных данных (не очень больших, естественно).

Экспериментальный раздел занятия

1. Даны натуральные числа n и k. Перечислить все последовательности длины n из чисел от 1 до k.

Количество последовательностей — k^n. Действительно, на каждое из n мест разрешается записывать числа от 1 до k, перемножаем и получаем результат. Строгое доказательство осуществляется традиционно, с помощью простой индуктивной схемы.

Пусть n=3, k=2 и последовательности выводятся в следующем порядке: 1 1 1, 1 1 2, 1 2 1, 1 2 2, 2 1 1, 2 1 2, 2 2 1, 2 2 2. Чем он (порядок) характеризуется? Возьмем две соседние последовательности, например 1 1 2 и 1 2 1. До какой-то позиции они совпадают? В данном случае до второй (несовпадение возможно и в первой позиции), а в этой позиции число во второй последовательности больше, чем число в первой последовательности. Такой порядок называется лексикографическим. Как из очередной последовательности получать следующую в лексикографическом порядке? Идем справа. Находим первый элемент, неравный k. Увеличиваем его на единицу, а «хвост» последовательности максимально ухудшаем, то есть записываем 1. В последней последовательности сделать такое преобразование, естественно, не представляется возможным.

Самое время потренироваться на примерах. Как вы поняли постановку задачи?

Одномерный массив из n элементов — очевидная структура данных для хранения последовательности. Элементами мас-

сива являются целые числа в диапазоне от 1 до k. Явно отразим это в описании, вместо типа Integer воспользуемся ограниченным типом 1..k.

```
Type MyArray=Array[1..n] Of 1..k;
```

Определим «костяк» программы и очевидную процедуру вывода элементов массива.

```
{$R+}
Program My13_1;
 Const n=3;
 k=2;
 Type MyArray=Array[1..n] Of 0..k;
 Var A:MyArray;
 Procedure Print;
  Var i:Integer;
  Begin
   For i:=1 To n Do Write(A[i]:3);
   WriteLn;
  End;
 Procedure Solve(t:Integer);
  Begin
  End;
 Begin
  FillChar(A,SizeOf(A),0) {Элементам массива A
                    присваивается значение ноль};
  Solve(1);
 End.
```

Из первого вызова рекурсивной процедуры Solve возникает вопрос — что определяет параметр рекурсии t? Он определяет позицию в последовательности (номер элемента в массиве A). Следующий вызов — Solve(t+1), при значении t, большем n, необходимо выводить очередную последовательность, т. е. записать очередной элемент, определяемый значением управляющей переменной i, в позицию t.

```
Procedure Solve(t:Integer);
  Var i:Integer;
  Begin
   If t>n Then Print
    Else For i:=1 To k Do Begin
```

```
        A[t]:=i;
        Solve(t+1);
       End;
  End;
```

Измените программу так, чтобы последовательности выводились в следующей очередности: 2 2 2, 2 2 1, 2 1 2, 2 1 1, 1 2 2, 1 2 1, 1 1 2, 1 1 1.

Предположим, что значение k больше значения n. Измените программу. В последовательности чисел i-й элемент не должен превосходить i. Это простое изменение программы. А сейчас, при этом же предположении ($k>n$), требуется сгенерировать возрастающие последовательности чисел. На месте t в последовательности допускается запись чисел из интервала от $A[t-1]+1$ до $k-(n-t)$. Действительно, $t=n$, максимальное число в позиции n равно, естественно, k; $t=n-1$, максимальное число в позиции $n-1$ есть $k-1$ и так далее. Эта мысль приводит к следующему изменению процедуры Solve.

```
Procedure Solve(t:Integer);
  Var i:Integer;
  Begin
   If t>n Then Print
    Else For i:=A[t-1]+1 To k-n+t Do Begin
         A[t]:=i;
         Solve(t+1);
        End;
  End;
```

Однако только этим изменением в программе ограничиться нельзя. Первый вызов процедуры Solve(1) и описание типа:

```
Type MyArray=Array[1..n] Of 1..k;
```

приводит к знакомой ошибке `Error 201: Range check error` в строке `For i:=A[t-1]+1 To k-n+t Do;`.

Причина очевидна — выход индекса за пределы массива A. Изменим описание типа на:

```
Type MyArray=Array[0..n] Of 1..k;
```

Программа выдает результат, но содержит логическую неточность. При первом вызове Solve используется неопределенное значение $A[t-1]$, то есть единицу прибавляем неизвестно к

чему. При изменении первого вызова на `A[0]:=0; Solve(1);` возникает новая ошибка: `Error 76: Constant out of range` (значение константы выходит за допустимый диапазон). Еще одно изменение:

```
Type MyArray=Array[0..n] Of 0..k;
```

и можно в первом приближении закончить эксперименты с программой.

2. С понятием перестановки чисел мы познакомились на предыдущем занятии. Непишем рекурсивную процедуру генерации перестановок чисел от 1 до n.

В чем здесь суть рекурсии? Фиксируем на первом месте очередное число и генерируем все перестановки этого вида. При этом поступаем точно по такой же схеме. Фиксируем число на втором месте и генерируем все перестановки уже с фиксированными элементами на первом и втором местах.

Пусть n=3. Перестановки должны генерироваться в следующей последовательности: 1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 2 1, 3 1 2.

Пусть n=4. Вместо многоточия запишите генерируемые перестановки: 1 2 3 4, 1 2 4 3, 1 3 2 4, 1 3 4 2, 1 4 3 2, 1 4 2 3, 2 1 3 4, 2 1 4 3, 2 3 1 4, 2 3 4 1, ..., 4 1 2 3.

Необходимо осознать, что после генерации всех перестановок с фиксированным элементом в позиции t перестановка возвращается в исходное состояние и осуществляется новый обмен значениями элемента из позиции с номером t+1 и i+1.

```
Program My13_2; {Процедура Print совпадает
       с соответствующей процедурой из предыдущей
       программы.}
 Const n=4;
 Type MyArray=Array[1..n] Of Integer;
 Var A:MyArray;
    i:Integer;
 Procedure Swap(Var a,b:Integer);
  Var c:Integer;
  Begin
   c:=a;
   a:=b;
   b:=c;
  End;
 Procedure Solve(t:Integer);
  Var i:Integer;
```

```
 Begin
  If t>=n Then Print
   Else For i:=t+1 To n Do Begin
         Swap(A[t+1],A[i]);
         Solve(t+1);
         Swap(A[t+1],A[i]);
        End;
 End;
Begin
 For i:=1 To n Do A[i]:=i; {Первая перестановка -
                            1 2 3 ... n}
 Solve(0); {Первый вызов}
End.
```

Для простоты будем считать, что каждый рекурсивный вызов приводит к созданию «копии» процедуры. Это не так, но при этом предположении нам легче формулировать вопросы. Сколько «копий» процедуры Solve создается при n=3? Сколько раз создаются копии с t=2? Сколько выполняется лишних обменов типа $A[j] \leftrightarrow A[j]$? На рис. 1.9 приводится часть схемы вызовов процедуры Solve (при n=3). Для ответа на сформулированные вопросы закончите рис. 1.9.

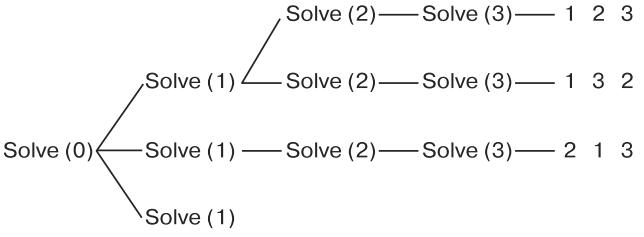


Рис. 1.9. Часть схемы вызовов процедуры Solve

Измените описание типа элементов у массива A на:

```
Type MyArray=Array[1..n] Of 1..n;
```

Возникает ошибка `Error 26: Type mismatch` при вызове процедуры Swap. Устраните ее.

3. Дано натуральное число n. Необходимо получить все разбиения числа n на сумму слагаемых $n=a_1+a_2+ \ldots +a_k$, где k, a_1, a_2, ..., $a_k>0$.

Будем считать разбиения эквивалентными, если суммы отличаются только порядком слагаемых. Множество эквивалентных сумм будем представлять убывающей последовательностью a_1, a_2, ..., a_k, где $a_1 \geq a_2 \geq \ldots \geq a_k$.

При $n=4$ разбиения 1+1+1+1, 2+1+1, 2+2, 3+1, 4 перечислены в лексикографическом порядке. Для каждой пары соседних разбиений находится номер позиции, в которой число из второго разбиения больше числа из первого разбиения, а до этой позиции последовательности совпадают. Будем генерировать разбиения в порядке, обратном лексикографическому: 4, 3+1, 2+2, 2+1+1, 1+1+1+1.

Рекурсивная схема реализации просматривается достаточно легко. Пусть мы записали в позицию t число a_t. Сумма чисел $a_1+a_2+\ldots+a_t=s$. С позиции $t+1$ мы генерируем все разбиения числа $n-s$, причем для элемента $A[t+1]$ (и остальных) должно выполняться неравенство $A[t+1] \leq A[t]$. Переход к следующему разбиению в позиции t сводится, если это возможно, к вычитанию единицы.

Уточним некоторые технические детали. Разбиение хранится в глобальном массиве A. Количество элементов в разбиении различно. Значение переменной t определяет текущий размер выводимой части массива A. Процедура Print претерпит небольшое изменение.

```
Procedure Print(t:Integer);
  Var i:Integer;
  Begin
   For i:=1 To t Do Write(A[i]:3);
   WriteLn;
  End;
```

Рекурсивная процедура Solve имеет два параметра: число n и позицию t, начиная с которой необходимо получить все разбиения числа n. Первым разбиением является разбиение, соответствующее записи числа n в позицию t, а затем ... Последовательно записываем в позицию t числа $n-1$, $n-2$, ... и так до 1, а с позиции $t+1$ генерируем все разбиения чисел 1, 2, ..., $n-1$ соответственно. Приведем текст решения.

```
{$R+}
Program My13_3;
 Const n=4;
```

```
Type MyArray=Array[1..n] Of Integer;
Var A:MyArray;
Procedure Solve(n,t:Integer);
 Var i:Integer;
 Begin
  If n=1 Then Begin A[t]:=1;Print(t);End
   Else Begin A[t]:=n;Print(t);
        For i:=1 To n-1 Do Begin
        A[t]:=n-i;
        Solve(i,t+1);
       End;
  End;
 End;
Begin
 Solve(n,1);
 ReadLn;
End.
```

Однако после запуска программы получается результат, несколько отличный от предполагаемого: 4, 3 1, 2 2, 2 1 1, 1 3, 1 2 1, 1 1 2, 1 1 1 1. Разбиения 1 3, 1 2 1 и 1 1 2 — лишние. Изменим процедуру.

```
Procedure Solve(n,t:Integer);
 Var i:Integer;
 Begin
  If n=1 Then Begin
          A[t]:=1;
          Print(t);
         End
   Else Begin
     If n<=A[t-1] Then Begin
       A[t]:=n;
       Print(t);
      End;
     For i:=1 To n-1 Do Begin
       A[t]:=n-i;
       Solve(i,t+1);
      End;
    End;
End;
```

Появление индекса $t-1$ требует изменить описание типа массива на:

```
Type MyArray=Array[0..n] Of Integer;
```

и выполнять начальное присвоение `A[0]:=n` перед первым вызовом Solve (n,1). Если этого не сделать, возникает ошибка `Error 201: Range check error`. Что мы пытались сделать? Отсечь лишние разбиения при их выводе, но не генерации! Запуск программы нас очередной раз разочарует. Результат работы: 4, 3 1, 2 2, 2 1 1, 1 2 1, 1 1 1 1. Изменим процедуру.

```
Procedure Solve(n,t:Integer);
  Var i,q:Integer;
  Begin
   If n=1 Then Begin
           A[t]:=1;
           Print(t);
          End
    Else Begin
      If n<=A[t-1] Then Begin
        A[t]:=n;
        Print(t);
       End;
      q:=Min(n-1,A[t-1]);
      For i:=q DownTo 1 Do Begin
        A[t]:=i;
        Solve(n-i,t+1);
       End;
     End;
  End;
```

Нам потребуется функция Min:

```
Function Min(a,b:Integer):Integer;
  Begin
   If a<b Then Min:=a Else Min:=b;
  End;
```

Получаем правильный результат: 4, 3 1, 2 2, 2 1 1, 1 1 1 1, но часть решений мы по-прежнему «отсекаем» только на выходе. Продолжите исследование. Научитесь выводить разбиения в следующих порядках:

```
1111,    211,    22,    31,    4;
1111,    112,    13,    22,    4;
4,       22,     13,    112,   1111.
```

Задания для самостоятельной работы

1. Написать рекурсивную программу вывода на экран картинки, изображенной на рис. 1.10.

1111111111111111	(16 раз)
222222222222	(12 раз)
33333333	(8 раз)
4444	(4 раза)
33333333	(8 раз)
222222222222	(12 раз)
1111111111111111	(16 раз)

Рис. 1.10. Вид данных на экране

2. В конце XIX века в Европе появилась игра под названием «Ханойские башни». Реквизит игры состоит из 3 игл, на которых размещается башня из дисков с отверстиями в центре. Цель игры — перенести башню с левой иглы (1) на правую (3), причем за один раз можно переносить только один диск, кроме того, запрещается помещать больший диск над меньшим.

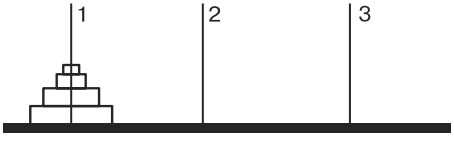


Рис. 1.11. Исходная ситуация в игре «Ханойские башни»

Выполните трассировку приведенного ниже решения с использованием отладчика системы программирования. Убедитесь в правильности решения. Напишите нерекурсивную реализацию программы решения задачи.

```
Program My13_z1;
 Var n: Integer;
```

```
Procedure MoveTown(High,FromI,ToI,WorkI:
                   Integer);
 Begin
  If High>0 Then Begin
   MoveTown(High-1,FromI,WorkI,ToI);
   WriteLn('Перенести кольцо №', High,
           ' с иглы ', FromI, ' на иглу ', ToI);
   MoveTown(High-1,WorkI,ToI,FromI);
  End;
 End;
Begin
 Write('Количество колец =');
 ReadLn(n);
 MoveTown(n,1,3,2);
End.
```

Данная задача обязана своим происхождением индийской легенде: в большом храме Бенареса бронзовая плита поддерживает три алмазных стержня, на один из которых бог нанизал при сотворении мира 64 золотых диска. С тех пор день и ночь монахи, сменяя друг друга, каждую секунду перекладывают по одному диску согласно описанным выше правилам. Конец мира наступит тогда, когда все 64 диска будут перемещены, на что потребуется чуть больше 58 миллиардов лет[10].

3. Составить рекурсивную функцию вычисления значения функции Аккермана для неотрицательных чисел n и m, вводимых с клавиатуры.

$$A(n,m)=\begin{cases} m+1 & \text{при } n=0; \\ A(n-1,1) & \text{при } n\neq 0,\ m=0; \\ A(n-1,A(n,m-1)) & \text{при } n>0,\ m>0. \end{cases}$$

где $A(n,m)$ — размещения без повторений.

Найдите одну пару значений n и m, при которых возникнет переполнение стека адресов возврата.

4. Функция $F(n)$ определена для целых положительных чисел следующим образом:

[10] Подробнее о задаче можно прочитать в книге: Окулов С. М., Лялин А. В. Ханойские башни. (серия: Развитие интеллекта школьника). — М.: БИНОМ. Лаборатория знаний, 2008.

$$F(n)=\begin{cases}1 & \text{при } n=1;\\ \sum\limits_{i=2}^{n} F(n \text{ Div } i) & \text{при } n\geq 2.\end{cases}$$

Соответствует ли приведенная ниже функция данному определению?

```
Function F(n:Integer):Integer;
 Var i,s:Integer;
 Begin
  If n=1 Then s:=1
   Else For i:=2 To n Do s:=s+F(n Div 2);
  F:=s;
 End;
```

Можно ли вычислить с помощью этой реализации функции значения F при n=5, 6, 7, ..., 20?

5. Число сочетаний $C(n,m)$ вычисляется по формуле $C(n,m)=C(n-1,\ m-1)+C(n-1,\ m)$, при этом $C(n,0)=1$ и $C(n,\ n)=1$. Напишите рекурсивную функцию для подсчета числа сочетаний. Определите область ее применения (диапазон допустимых значений n и m).

Занятие № 14. Символьный и строковый типы данных

План занятия

1. Символьный тип данных.
2. Короткие программы — иллюстрации работы с символьным типом данных.
3. Строковый тип данных.
4. Стандартные процедуры и функции работы с данными строкового типа.
5. Эксперименты с программами (выделение слов из текста; вставка символа пробела после запятой; нахождение расстояния между строками; генерация всех подстрок строки; определение эквивалентности двух слов).
6. Выполнение самостоятельной работы.

Символьный тип данных

Значениями типа Char являются символы. Символьные константы заключаются в апострофы, например: '!', '3', '*G*'. Всего имеется 256 различных символов (замечание для профессионалов: мы не рассматриваем кодировки, отличные от восьмибитовых, и говорим только о типе Char в Паскале).

Все множество символов образует **таблицу символов**. Символы в таблице пронумерованы целыми числами от 0 до 255. Номер символа в таблице называется его **кодом**. Вообще говоря, имеется много таблиц символов (кодировок), но мы будем использовать одну — наиболее распространенную — таблицу ASCII (American Standard Code for Information Interchange — Американский стандартный код для обмена информацией).

Таблица ASCII состоит из двух частей: первая половина (символы с кодами от 0 до 127) стандартизована и одинакова для всех компьютеров, на которых используется данная кодировка; в первую половину таблицы входят цифры, знаки препинания, буквы латинского алфавита и другие общеупотребляемые символы. Вторая половина таблицы (символы с кодами от 127 до 255) содержит символы того или иного национального алфавита (в нашем случае — русского), символы псевдографики и так далее.

Символы	Коды десятичные	Коды двоичные
Цифры 0–9	48–57	00110000–00111001
Буквы A–Z	65–90	01000001–01011010
Буквы a–z	97–122	01100001–01111010
Буквы А–Я	128–159	10000000–10011111

Для получения кода символа и наоборот в языке Паскаль существуют две функции: Ord и Chr. Функция Ord(w) дает порядковый номер символа w, Chr(i) определяет символ с порядковым номером i. Функции Ord и Chr обратны по отношению друг к другу:

Chr(Ord(w))=w;

Ord(Chr(i))=i.

Отношение порядка на множестве символов позволяет выполнять операции сравнения. Из двух символов «меньше» тот, который встречается раньше в кодировке ASCII. Для величин типа Char функции Pred и Succ работают следующим образом:

Pred(q)=Chr(Ord(q)–1);

Succ(q)=Chr(Ord(q)+1).

Короткие программы

1. В первом примере выводятся символы и соответствующие им коды. Переменная *k* используется как счетчик для организации последовательного вывода — по 5 символов.

```
Program My14_1;
 Var i,k:Integer;
 Begin
  WriteLn('Вывод порядковых номеров (кодов)
          символов - значение переменной i и самих
          символов.');
  For i:=1 To 255 Do Begin
   Write(i:4,' Символ ',Chr(i));
   k:=k+1;
   If k=5 Then Begin
    WriteLn;
    k:=0;
   End;
  End;
 End;
```

2. Во втором примере показано, как переменная символьного типа используется в качестве управляющей переменной в операторе For. Символы выводятся так:

```
A
BB
CCC
...
WWW...WWW  (23 раза)
```

```
Program My14_2;
 Var i:Char;
     j:Integer;
 Begin
  For i:='A' To 'W' Do Begin
   For j:=1 To Ord(i)-Ord('A')+1 Do Write(i);
   WriteLn;
  End;
 End.
```

Определите, что будет выводиться на экран в результате следующей не очень значительной модификации программы.

```
Program My14_2m;
 Var i,j:Char;
 Begin
  For i:='a' To 'z' Do
   For j:='a' To i Do Write(j);
  ReadLn;
 End.
```

3. В следующем примере подсчитывается количество символов, введенных с клавиатуры. Ввод заканчивается символом '.'. Введите несколько символов, затем точку и нажмите клавишу Enter. Программа выдаст правильный результат. А если нажимать клавишу Enter после ввода каждого символа? Результат явно неверный, он как будто бы в три раза превышает ожидаемый истинный результат. На самом деле все верно. Нажатие клавиши Enter генерирует ввод еще двух символов (управляющих) — возврата каретки (код 13) и перевода строки (код 10), и поскольку точка после каждого символа не ставилась, эти управляющие символы каждый раз воспринимались как подлежащие счету.

```
Program My14_3;
 Var i:Char;
     j:Integer;
 Begin
  Read(i);
  j:=0;
  While i<>'.' Do Begin
    j:=j+1;
    Read(i);
   End;
  WriteLn(j);
 End.
```

Другая версия этой простой программы позволяет отказаться от символа «точка» как признака конца ввода данных.

```
Program My14_3m;
 Var i:Char;
     j:Integer;
 Begin
  Read(i);
```

```
 j:=0;
 While i<>#13 Do Begin
   j:=j+1;
   Read(i);
  End;
 WriteLn(j);
 ReadLn;
End.
```

Обратите внимание на запись символа с кодом 13: #13. Это еще один способ записи символьных констант. А если изменить цикл на следующий:

```
While i<>#10 Do Begin j:=j+1;Read(i);End;
```

то количество подсчитанных символов будет на единицу больше. Объясните результат.

4. В следующем примере подсчитывается количество цифр в вводимых с клавиатуры данных.

```
Program My14_4;
 Var   ch:Char;
       k:Integer;
 Begin
  Read(ch);
  k:=0;
  While ch<>#13   Do Begin
     If (ch>='0') And (ch<='9') Then k:=k+1;
    Read(ch);
   End;
  WriteLn('Количество цифр равно ',k);
End.
```

Измените программу так, чтобы она:

- определяла, являются ли введенные данные правильной записью целого числа;
- вычисляла сумму цифр введенного числа.

Функция Upcase(*ch*) используется для преобразования строчных латинских букв в прописные.

Определите, какая обработка вводимых символов выполняется в следующем примере.

```
Program My14_4m;
 Var  ch:Char;
```

```
Begin
 Read(ch);
 While ch<>#13 Do Begin
   If (ch>='a') And (ch<='z')
    Then Write(Upcase(ch))
    Else Write(ch);
   Read(ch);
 End;
End.
```

Строковый тип данных

Строка представляет собой последовательность символов определенной длины. Для обозначения строкового типа используется ключевое слово String.

Примеры описания переменных типа String:

```
Var  Str1:  String[10];
     Str2:  String;
     Str3: String[13].
```

В квадратных скобках указывается максимальный размер (длина) строки. Если он не указан, то длина строки считается равной 255 символам. Заметим, что строку можно рассматривать как одномерный массив символов — к любому символу строки допустимо обращение по его номеру (*Str*1[*i*] — это обращение к *i*-му элементу строки *Str*1). Первый символ строки (с номером 0) содержит фактическую длину строки. Переменные типа String выводятся на экран монитора посредством стандартных процедур Write и WriteLn и вводятся с помощью ReadLn и Read, т. е. вводятся и выводятся не поэлементно, как массивы, а сразу целиком. Следующий простой пример иллюстрирует сказанное.

```
Program My14_5;
 Var s:String;
     w:String[10];
     v:String[5];
     i,j:Integer;
 Begin
  ReadLn(v); WriteLn(v); WriteLn(Integer(v[0]));
  ReadLn(w); WriteLn(w); WriteLn(Ord(w[0]));
```

```
  ReadLn(s); WriteLn(s); WriteLn(Integer(s[0]));
  For i:=1 To Ord(s[0]) Do Begin
   For j:=1 To i-1 Do Write(' ');
   WriteLn(s[i]);
  End;
  ReadLn;
 End.
```

Если ввести строки *v* и *w* большей длины, чем указано в описании, то они «обрезаются».

Операторы WriteLn(Integer(*v*[0])) и WriteLn(Ord(*w*[0])) обеспечивают вывод значения длины строки. Если изменить первый оператор на WriteLn(*v*[0]), то вместо цифрового значения на экран выводится «непонятный» символ. Попробуйте объяснить этот результат и понять смысл преобразования Integer(*v*[0]).

Последующие операторы этого примера демонстрируют обращение к элементам строки. Вывод символов строки *s* на экран осуществляется «лесенкой».

Сравнение строк происходит посимвольно слева направо: сравниваются коды соответствующих символов до тех пор, пока не нарушится равенство, при этом сразу делается вывод о знаке неравенства. Две строки называются равными, если они равны по длине и совпадают посимвольно.

Примеры:

```
'Balkon'<'balkon'    (Ord('B')<Ord('b'));
'balkon'>'balken'    (Ord('o')>Ord('e'));
```

`'balkon'>'balk'` (длина первой строки больше);

`'кошка  '>'кошка'` (длина первой строки больше);

`'Кот'='Кот'` (строки равны по длине и совпадают посимвольно).

Строковая константа, так же как и отдельные символы, заключается в апострофы. Например:

```
Str1:='У Егорки'; Str2:='всегда отговорки';.
```

Таблица 1.19

Стандартные процедуры и функции работы со строковым типом данных

Тип	Вызов	Параметры	Действие
Процедура	`Delete(s,p,n)`	`Var s:String; p,n:Integer`	Из строки *s*, начиная с позиции *p*, удаляются *n* символов
Процедура	`Insert(w,s,p)`	`w:String; Var s:String; p:Integer;`	В строку *s*, начиная с позиции *p*, вставляется строка *w*
Процедура	`Str(v,s)`	`v:Integer` или `v:Real; Var s:String;`	Число *v* преобразуется в строку, результат в *s*
Процедура	`Val(s,v,w)`	`s:String; Var v:Integer` или `Var v:Real; Var w:Integer;`	Если строка *s* состоит из цифр, то они преобразуются в числовое значение переменной *v*, значение *w* равно 0. Если строка состоит не только из цифр, преобразование не выполняется, *w*<>0 — признак ошибки
Функция	`Concat(s1,s2,…,sm)` или `s1+s2+…+sm` Тип значения функции — `String`	`s1,s2, …, sm: String;`	Строки *s*1+*s*2+…+*sm* записываются одна за другой. Если результат превысит 255 символов, строка обрывается
Функция	`Copy (s, p, n)` Тип значения функции — `String`	`s:String; p,n:Integer;`	Из строки *s*, начиная с позиции *p*, выбирается *n* символов
Функция	`Length(s)` Тип значения функции — `Integer`	`s:String;`	Определяется длина строки *s*, то есть число символов, из которых она состоит
Функция	`Pos(w,s)` Тип значения функции — `Integer`	`w,s:String;`	В строке *s* отыскивается первое вхождение строки *w* (номер позиции). Если вхождения нет, то выдается 0

Примеры:

В приведенных примерах переменные *s*1, *s*2, *s*3 имеют тип String, переменные *p*, *q* — тип Integer.

1. Операторы:

```
s1:='У Егорки всегда отговорки';
Delete(s1,7,8);
```

Результат: *s*1=’У Егора отговорки’.

2. Операторы:

```
s1:='У Егорки всегда отговорки';
s2:='Матрёны и ';
Insert(s2,s1,3);
```

Результат: *s*1=’У Матрёны и Егорки всегда отговорки’.

3. Операторы:

```
p:=1234;q:=34.5;
Str(p,s1); Str(q,s2);
```

Результат: *s*1=’1234’, *s*2=’34.5’.

4. Операторы:

```
s1:='555';s2:='23.345';s3:='34rr2';
Val(s1,p,w); Val(s2,q,w); Val(s3,p,w);
```

Результат:

- в первом случае *p*=555, *w*=0;
- во втором случае *q*=23.345, *w*=0;
- в третьем случае *w*<>0, *p*=???.

5. Операторы:

```
s1:='У Егорки всегда отговорки, ';
s2:='у Миладки всегда шоколадки';
s3:=Concat(s1,s2); (или s3:=s1+s2;)
```

Результат: *s*3=’У Егорки всегда отговорки, у Миладки всегда шоколадки’.

6. Операторы:

```
s1:='У Егорки всегда отговорки, у Миладки всегда
шоколадки'
s2:=Copy(s1,28,26);
```

Результат: *s*2=’у Миладки всегда шоколадки’.

7. Операторы:

```
s1:='У Егорки всегда отговорки';
p:=Length(s);
```

Результат: *p*=25.

8. Операторы:

```
s1:='У Егорки всегда отговорки';
p:=Pos('o',s);
```

Результат: p=5.

Продолжите примеры. Напишите программу для исследования работы перечисленных процедур и функций. Обратите особое внимание на граничные условия (например, выясните, что получается, если длина результата больше 255). В этой работе можно использовать приведенные ниже короткие примеры.

1. Подсчитать количество символов (параметр q) в строке (параметр *st*):

```
Function QChar(q:Char;st: String): Byte;
 Var i, k:  Byte;
 Begin
  k:=0;
  For  i:=1 To Length(st) Do
   If st[i]=q Then k:=k+1;
  QChar:=k;
 End;
```

2. Удалить среднюю букву при нечетной длине строки и две средние буквы при четной длине строки:

```
Procedure MiDel( Var st: String);
 Var k: Byte;
 Begin
  k:=Length(st);
  If k Mod 2=1 Then Delete(st, k Div 2+1,1)
   Else Delete(st,k Div 2,2);
 End;
```

3. Заменить все вхождения подстроки w в строке *st* на подстроку v:

```
Procedure Ins(w,v:String;Var st: String);
 Var k: Byte;
 Begin
  While Pos(w,st)<>0 Do Begin
   k:=Pos(w,st);
   Delete(st,k,Length(w));
   Insert(v,st,k);
  End;
 End;
```

4. Подсчитать сумму цифр, встречающихся в строке:

```
Function Sum(st: String): Integer;
 Var  i, k, d, s: Integer;
 Begin
  s:=0;
  For i:=1 To Length(st) Do Begin
   Val(st[i],d,k);
   If k=0 Then s:=s+d;
  End;
  Sum:=s;
 End;
```

Экспериментальный раздел занятия

1. Дана строка. Считаем ее отрывком текста. Группы символов, разделенных одним или несколькими пробелами, назовем словами. Пробелы могут находиться как в начале текста, так и в конце.

Требуется выделить слова из текста и каждое слово записать в соответствующий элемент массива.

Приведенная ниже программа решает эту задачу. Выполните ее в режиме отладчика, наблюдая за изменением значений переменных.

```
Program My14_6;
 Const n=20; m=10; {Количество слов в тексте
      и количество букв в слове. Естественно,
      что эти параметры программы допускается
      изменять.}
 Type TString=String[m];
 Var A:Array[1..n] Of TString;
     s:String;
     k,i:Integer;
 Procedure DelPr(Var s:String); {Удаляем пробелы
      в начале текста.}
  Begin
   While (s[1]=' ') And (s<>'') Do Delete(s,1,1);
  End;
 Function GetWord(Var s:String):TString;
       {Выделяем слово, при этом оно удаляется
       из текста, и убираем пробелы после слова.}
  Begin
```

```
  GetWord:=Copy(s,1,Pos(' ',s)-1);
  Delete(s,1,Pos(' ',s));
  DelPr(s);
 End;
Begin
 WriteLn('Введите текст');
 ReadLn(s);
 s:=s+' '; {Добавляем символ пробела в конец
           текста. Зачем?}
 DelPr(s); {Удаляем пробелы в начале текста.}
 k:=0;
 While s<>'' Do Begin {Пока текст не пустой.}
   k:=k+1;
   A[k]:=GetWord(s); {Берем слово из текста.}
  End;
 For i:=1 To k Do WriteLn(A[i]);
 ReadLn;
End.
```

Удалите вызов процедуры DelPr(*s*) из основной программы, а в функции GetWord переставьте этот вызов в ее начало. Что изменится в работе программы? Для каких исходных данных она не будет правильно работать? Что произойдет, если в конец текста не добавлять символ пробела? В процедуре DelPr измените

```
While (s[1]=' ') And (s<>'') Do Delete(s,1,1);
```

на

```
While (s[1]=' ') Do Delete(s,1,1).
```

Приведите пример исходного текста, при котором результат работы программы окажется неверным. Продолжите эксперименты с программой.

2. По правилам машинописи после запятой в тексте всегда ставится пробел. Следующая программа вставляет недостающие пробелы:

```
Program My14_7;
 Var i:Integer;
     s:String;
 Begin
  WriteLn('Введите текст');
  ReadLn(s);
```

```
 i:=1;
 While i<Length(s) Do Begin
  If (s[i]=',') And Not(s[i+1]=' ')
   Then Insert(' ',s,i+1);
  i:=i+1;
 End;
 WriteLn(s);
 ReadLn;
End.
```

Почему для организации цикла выбран оператор While, а не For? Почему в теле цикла нельзя использовать функцию Pos(' ',*s*)? Обратите внимание: если запятая — последний символ текста, то мы не добавляем пробела. Измените программу для исправления случаев нарушения правила «после символов '!', '?', '.' должен стоять пробел, а затем текст начинается с прописной буквы». Вспомните еще ряд традиционных ошибок и модифицируйте программу для их исправления.

3. Даны две строки *X* и *Y*. Назовем расстоянием *r* между *X* и *Y* количество символов, которыми *X* и *Y* различаются между собой (то есть нас интересует минимальное количество символов, которые необходимо добавить в строки *X* и *Y* для того, чтобы после добавления они состояли из одних и тех же символов; при этом существенно количество символов, а не их порядок).

Например:

X='*abcd*', *Y*='*dxxc*', *r*=4;

X='1111111', *Y*='111222', *r*=7.

Программа вычисления расстояния между строками выглядит так:

```
Program My14_8;
 Var i,j,r:Integer;
     S,X,Y:String;
 Begin
  WriteLn('Первая строка');
  ReadLn(X);
  WriteLn('Вторая строка');
  ReadLn(Y);
  If Length(X)>Length(Y) Then Begin
    S:=X;
    X:=Y;
```

```
    Y:=S;
   End; {Y строка большей длины.}
  r:=Length(Y);
  For i:=1 To Length(X) Do Begin
   j:=1;
   While (j<=Length(Y)) And (Y[j]<>X[i]) Do j:=j+1;
   If j>Length(Y) Then r:=r+1
    Else Begin
      r:=r-1;
      Delete(Y,j,1);
     End; {Есть совпадение. Уменьшаем расстояние
           и удаляем символ из Y.}
  End;
  WriteLn('Расстояние ',r);
  ReadLn;
 End.
```

К каким последствиям приведет исключение из решения строки со сравнением длин строк и записи в *Y* строки наибольшей длины? Проверьте.

Исключим оператор Delete(*Y*,*j*,1). Какой результат у нас получится?

Измените программу так, чтобы в тексте находились два слова, расстояние между которыми имеет максимальное (минимальное) значение.

4. Дана строка не более чем из шести произвольных различных символов. Разработать программу вывода всех возможных подмножеств (подстрок), составленных из символов данной строки. Количество подстрок равно 2^n-1, где n — количество символов в строке. Естественно, что каждый символ входит в подстроку не более одного раза. Будем генерировать шестиразрядные двоичные числа. Так, числу 010101 соответствует подстрока 'bdf' строки 'abcdef'. Для получения очередного числа просматриваем число слева направо до первого нуля, заменяя при этом все 1 на 0, затем, встретив 0, заменяем его на 1. Например, после 110011 получаем 001011. Завершение генерации — число 0000001.

```
Program My14_9;
 Const MaxN=6;
 Type MyArray=Array[1..MaxN+1] Of Byte;
 Var A,Last:MyArray;
```

```
    i:Integer;
    s:String;
Function Eq(X,Y:MyArray;n:Integer):Boolean;
        {Сравнение текущего значения с 0000001,
         если n=6.}
 Var i:Integer;
 Begin
  i:=1;
  While (i<=n) And (X[i]=Y[i]) Do i:=i+1;
  Eq:=(i>n);
 End;
Begin
 WriteLn('Строка ');
 ReadLn(s);
 FillChar(A,SizeOf(A),0);
 FillChar(Last,SizeOf(Last),0);
 Last[Length(s)+1]:=1; {Первое несуществующее
       разбиение.}
 While Not(Eq(A,Last,(Length(s)+1))) Do Begin
        {Пока нет совпадения, выполняем генерацию
        следующей подстроки.}
  i:=1;
  While (i<=Length(s)) And (A[i]=1) Do Begin
    A[i]:=0;
    i:=i+1;
   End;
  A[i]:=1;
  If i<=Length(s) Then
   For i:=1 To Length(s) Do
    If A[i]=1 Then Write(S[i]); {Вывод подстроки.}
  WriteLn;
 End;
 ReadLn;
End.
```

В схеме генерации реализовано обычное прибавление единицы к двоичному числу (только число записывается слева направо: 000000, 100000, 010000, 110000, 001000 и так далее). Каждому такому числу ставится в соответствие подстрока. Измените схему генерации так, чтобы единица прибавлялась традиционным способом: 000000, 000001, 000010, 000011, 000100 и так далее.

А можно ли исключить генерацию последовательности двоичных чисел из решения задачи?

5. Рассмотрим следующую задачу Заданы две строки A и B длиной n ($1 \le n \le 100$), состоящие из символов 0 и 1 (двоичные слова). Допускается следующее преобразование строк. Любую подстроку A или B, содержащую четное число единиц, можно перевернуть, то есть записать в обратном порядке. Например, в строке 11010100 можно перевернуть подстроку, составленную из символов с 3-й по 6-ю позиции. Получится строка 11101000. Две строки считаются эквивалентными, если одну из них можно получить из другой с помощью описанных преобразований. Определить эквивалентность заданных строк A и B; если строки эквивалентны, предложить один из возможных способов преобразования строки A в строку B.

Пример:

Даны строки 100011100, 001011001. Решить для них поставленную выше задачу. Ответ — «да»; искомые преобразования: 6 9, 3 8, 1 5.

Основная идея решения задачи формулируется следующим образом. На множестве всех слов W следует выделить подмножество S ($S \subset W$), такое что для любого слова $t \in W$ можно указать эквивалентное ему слово $r \in S$. Алфавит и слова у нас заданы, преобразования тоже. Осталось определить множество S, свести каждое слово к его представителю из множества S, и если представители совпадают, то слова эквивалентны. Из каких слов состоит S? Слов вида 11...100...0100...0, причем как первая группа единиц, так и первая группа нулей, а также следующая единица (слово из одних 0) могут отсутствовать. Основные фрагменты решения.

```
Const Max=100;
Type Tstr=String[Max];
....
Procedure Reverse(Var S:Tstr;i,j:Integer);
   {Достаточно очевидная процедура - перевертываем
    подстроку S с позиции i до позиции j.}
 Var ch:Char;
 Begin
  While i<j Do Begin
    ch:=S[i];
    S[i]:=S[j];
```

```
    S[j]:=ch;
    i:=i+1;
    j:=j-1;
  End;
End;
Procedure Normalize(Var S:Tstr); {Сводим слово
       к его представителю из S.}
Var i,j,cnt:Integer;
Begin
     cnt:=0;
     For i:=1 To Length(S) Do
     If S[i]='1' Then cnt:=cnt+1; {Подсчитываем
       количество единиц в слове.}
     For i:=1 To cnt-1 Do
      If S[i]='0' Then Begin {Находим «левый» 0
        в слове.}
      j:=i;
      Repeat j:=j+1 Until S[j]='1'; {Поиск первой
        единицы.}
      Repeat j:=j+1 Until S[j]='1'; {Поиск второй
        единицы, подстрока содержит четное число
        единиц, ее можно «перевернуть».}
      Reverse(S,i,j);
    End;
End;
```

Допишите программу. Заметим, что вывод преобразований, в случае эквивалентности слов, требует дополнительной структуры данных (массива) для запоминания значений i и j при каждом перевертывании.

Если исключить из условия задачи вывод преобразований, то ее можно решить гораздо проще. Рассмотрим табл. 1.20 (мы пока специально поставили знаки вопроса в заголовках двух столбцов).

Таблица 1.20

№	Слово	Количество единиц	?	?
1	1111000010	5	4	1
2	11111001	6	0	2
3	0000001000	1	6	3
4	10010001100	4	5	2

Окончание таблицы 1.20

№	Слово	Количество единиц	?	?
5	11000100100	4	5	2
6	11100100000	4	5	2
7	0100101101	5	3	2
8	1001001101	5	3	2
9	1100100101	5	3	2
10	1110010001	5	3	2
11	1111000100	5	3	2

Внимательно изучив таблицу, можно предположить, что в последних столбцах указано число нулей. Но каких? В четвертом столбце — количество нулей после четного числа единиц, в пятом — после нечетного. И вторая часть нашего вывода: эти числа не изменяются в процессе преобразования слов (строки таблицы с номерами 4–6 и 7–11). После этого можно написать небольшой фрагмент программы подсчета этих чисел (сразу после их ввода):

```
...
cnt:=0;
ch1:=0;
ch2:=0;
For I:=1 To Length(S) Do
If S[i]='0' Then
            If cnt Mod 2=0 Then ch1:=ch1+1
             Else ch2:=ch2+1
  Else cnt:=cnt+1;
...
```

Итак, если значения $ch1$ и $ch2$ для слов совпадают, то они эквивалентны в вышеприведенном значении. Нулевая позиция в слове считается при этом четной, например числа 1, 5, 4 однозначно определяют слово 0000010000.

Обоснуем вывод. Инвариантом преобразования является знакочередующая сумма вида $\sum(-1)^i \cdot a_i$, где i — номер единицы, a_i — номер позиции i-й единицы.

Обозначим через s номер позиции в подстроке, соответствующий центру переворачиваемой подстроки. Пусть x — номер позиции до «переворота», y — номер этой позиции после операции «переворота». Очевидно, что $(x+y)/2=s$, то есть $y=2s-x$ или $a_i \to 2s-a_i$.

Количество элементов в сумме четно, знаки чередуются, и $\sum(-1)^i \cdot a_i = \sum(-1)^i \cdot (2s - a_i)$.

Поэтому после ввода слов следует подсчитать количество единиц в каждом из них и суммы указанного типа. Если они совпадают, то слова A и B эквивалентны.

Например, для строки 0101 сумма равна $(-1)^1 \cdot 2 + (-1)^2 \cdot 4 = 2$, после переворачивания имеем 1010, сумма равна $(-1)^1 \cdot 1 + (-1)^2 \cdot 3 = 2$. Суммы совпадают, строки эквивалентны. Для строк 110100 и 110001 эти суммы равны –3 и –5, строки неэквивалентны.

Задания для самостоятельной работы

1. Напишите программу вывода последовательности символов:
   - `ZYYXXX...AA..AA`;
   - `ABC...ZZBC...ZZZC...ZZ..ZZ`.
2. Напишите программу, которая выводит True, если в строке буква A встречается чаще, чем буква B, и False в противном случае.
3. Проверьте, правильно ли в заданном тексте расставлены круглые скобки (то есть находится ли справа от каждой открывающей скобки соответствующая ей закрывающая скобка, а слева от каждой закрывающей — соответствующая ей открывающая).
4. Подсчитайте количество гласных латинских букв в строке.
5. Удвойте вхождение некоторой буквы в текст. Например, текст «мама папа» должен иметь вид — «маамаа паапаа».
6. Даны две строки. Выведите буквы, встречающиеся и в той, и в другой строке.
7. Дан текст. Выведите все слова, начинающиеся с прописных букв латинского алфавита.
8. Дан текст. Определите:
   - длину самого короткого и самого длинного слов;
   - количество слов, начинающихся и оканчивающихся одной и той же буквой;
   - количество слов, в которых содержится хотя бы одна заданная буква;
   - количество слов, в которые заданная буква входит n раз;
   - количество слов, являющихся палиндромами.

9. Дан текст. Выведите слова, встречающиеся в тексте по одному разу.

10. Дан текст. Выведите имеющиеся в нем различные слова.

11. Дан текст. Выведите все его слова, предварительно преобразовав каждое из них по следующему правилу:

- оставить в слове только первые вхождения каждой буквы;
- удалить из слова все предыдущие вхождения последней буквы.

12. Дан текст. Выведите слова, которые отличны от последнего слова и удовлетворяют следующим условиям:

- в слове нет повторяющихся букв;
- буквы слова упорядочены по алфавиту;
- слово совпадает с начальным отрезком латинского алфавита (a, ab, abc, abcd, ...).

13. Дан текст. Выведите все его слова, предварительно заменив в них первые буквы на заглавные.

14. Напишите программы решения ребусов:

ОДИН+ОДИН+ОДИН+ОДИН+ОДИН=ПЯТЬ;
КУБ=(К+У+Б)^3;
ТРИ+ДВА=ПЯТЬ;
VOLVO+FIAT=MOTOR.

При решении ребусов одинаковым буквам должны соответствовать одинаковые цифры, неважно какие именно — какие подойдут, поэтому ответов может быть несколько. Вот, например, часть ответов последнего ребуса (всего их 10): 15615+9743=25358, 15715+9643=25358, 72672+9451=82123.

15. Дан текст. Написать программу проверки правильности написания сочетаний «жи», «ши», «ча», «ща», «чу» и «щу». Исправить ошибки.

16. Даны три строки. Определите, можно ли из символов первых двух строк получить третью строку.

17. Даны две строки. Определите, можно ли, переставляя символы в первой строке, получить вторую строку.

Примечание

Термин **текст** в условиях задач следует понимать в смысле первого примера экспериментальной части данного занятия.

Занятие № 15. Текстовые файлы

План занятия

1. Файловый тип данных.
2. Процедуры и функции работы с файловым типом данных.
3. Короткие программы — иллюстрации работы с текстовыми файлами.
4. Эксперименты с программами (подсчет частоты записи целых чисел в файле; проверка правильности расстановки скобок в тексте программы; подсчет количества строк в тексте).
5. Выполнение самостоятельной работы.

Файловый тип данных

Данные, размещаемые программой в оперативной памяти компьютера, исчезают при выключении питания. Для долговременного хранения данных используются внешние носители, в частности магнитные диски. Единицей хранения информации на диске служит файл.

Файл имеет имя, и это имя относится к операционной системе, под управлением которой работает Паскаль, то есть никакого отношения к среде программирования не имеет.

С другой стороны, Паскаль должен как-то работать с файлом, поэтому файлу, как обычной переменной, в системе Паскаль присваивается дополнительное имя. Таким образом, в Паскале мы используем одно имя файла, а операционная система, фактически осуществляющая чтение-запись на диск, работает с другим именем. Естественно, существуют средства установления соответствия между этими именами.

Важно, что в конкретный момент времени с любым файлом можно работать или в режиме чтения, или в режиме записи. Одновременно и читать, и записывать нельзя. Для того чтобы, например, записать данные в файл, открытый для чтения, необходимо закончить работу с файлом в этом режиме, и затем начать работу с этим же файлом в режиме записи.

На этом занятия мы ограничимся работой с текстовыми (последовательными) файлами.

Информация в текстовых файлах хранится в виде последовательности символов. Символы составляют строки произвольной длины. В конце каждой строки записываются два особых символа: #13 #10 (соответственно возврат каретки и перевод строки), которые отделяют данную строку от следующей.

На самом деле с текстовыми файлами мы работали уже на первом занятии, не упоминая об этом явно. Существуют стандартные файлы ввода и вывода, связанные соответственно с клавиатурой и дисплеем. Имена этих файлов в системе Паскаль (файловые переменные) — Input и Output.

Таким образом, при выполнении операторов Read или Write фактически происходило чтение из стандартного файла ввода с клавиатуры или запись в стандартный файл вывода, связанный с дисплеем. Правда, можно возразить, что почти всегда мы вводим или выводим цифровую информацию, а здесь речь идет о текстовом файле. Оказывается, при чтении и записи чисел в текстовый файл (в частности, и в Input, и в Output) их преобразование в строковый тип выполняется автоматически.

Процедуры и функции для работы с файловым типом данных

В программе на Паскале текстовый файл представлен файловой переменной типа Text: Var <имя файловой переменной>: Text;.

Связь файловой переменной с дисковым именем файла устанавливается с помощью процедуры: Assign(<имя файловой переменной>, <дисковое имя файла>).

Второе имя больше нигде в программе не появляется.

Открытие файла для чтения выполняется процедурами:

- для чтения: Reset(<файловая переменная>);
- для записи: ReWrite(<файловая переменная>);
- для дозаписи: Append(<файловая переменная>).

Окончание работы с файлом фиксируется с помощью процедуры Close(<файловая переменная>).

Пример оформления работы с текстовым файлом:

```
Var f:Text;
    s:String;
Begin
 Assign(f,'a:/data.txt'); {Файл с именем data.txt
       находится на диске с логическим именем a.}
 Reset(f); {Файл открывается для чтения.}
 ReadLn(f,s); {Читаем одну строку из файла.}
...
 Close(f);
End.
```

Чтение из файла допускается лишь последовательное. Мы не можем вначале прочитать сотую строку, а затем десятую. Последовательный файл в этом отношении похож на ленту магнитофона или, если угодно, на линейку, по которой мы движемся слева направо и, чтобы попасть на десятый сантиметр, необходимо пройти все предыдущие.

Элементы данных файла отделяются друг от друга разделителями, ибо они могут быть разной длины. Кроме того, файл обязан заканчиваться каким-то признаком конца файла, иначе непонятно, когда закончить его чтение. В Паскале есть специальная функция Eof (<файловая переменная>), которая возвращает значение True, если достигнут конец файла, и False в противном случае.

Короткие примеры

Начнем с очень простого примера. Читаем данные (строки) из файла и выводим их на экран.

```
Program My15_1;
 Var f:Text;
     s:String;
Begin
  Assign(f,'Input.Txt');
  Reset(f);
  While Not Eof(f) Do Begin
   ReadLn(f,s);
   WriteLn(s);
  End;
  Close(f);
End.
```

При запуске программы возникает ошибка `Error 2: File not found`. Файла с именем Input.Txt нет на диске, его необходимо создать. Выбираем в текстовом редакторе Паскаля пункт **File/New** и записываем строки:

```
A
Bb
Ccc
Dddd
Eeeee
Ffffff
Ggggggg
```

и сохраняем файл с именем Input.Txt.

После повторного запуска программы мы видим на экране правильный результат, но только при условии, что наша программа и файл Input.Txt находятся в одном и том же каталоге.

Давайте немного поэкспериментируем. Измените текущий каталог — режим **File/Change Dir** — и выберите другой каталог. Запустите программу. Опять знакомая ошибка: `Error 2: File not found`.

Ваша программа находится в одном каталоге, а файл Input.Txt в другом. Перепишите файл Input.Txt в тот каталог, где находится ваша программа. Запустите программу, она должна работать нормально.

Измените программу. Вместо ReadLn(*f*,*s*) напишите Read(*f*,*s*). Теперь программа зацикливается, и приходится принудительно остановить ее работу, воспользовавшись клавишами Ctrl+Break. Причина — программа не может найти признак конца файла. Откройте окно **Debug/Watch**, введите имя переменной *s* и выполните программу в пошаговом режиме. Первую строку мы благополучно вводим и выводим на экран, а затем идут пустые строки. Попробуйте дать объяснение.

Если вам приходится все время использовать **F6** для того, чтобы перейти от одного окна к другому, значит, вы забыли материал занятия 3 (работу с окнами). Измените размеры окон и переместите их так, чтобы они не накладывались друг на друга. Например, их можно расположить так, как показано на рис. 1.12.

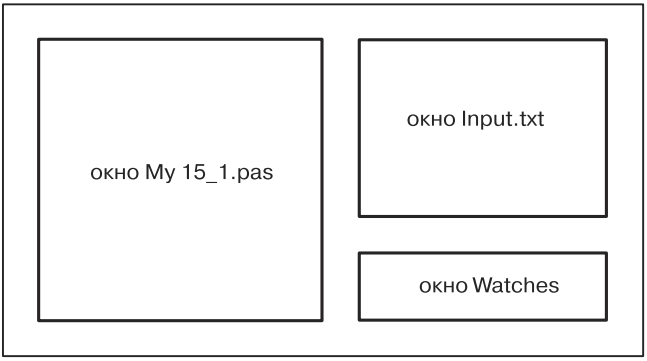


Рис. 1.12. Рациональное расположение окон на экране

А можно ли читать посимвольно? Изменим программу.

```
Program My15_1m;
 Var f:Text;
     ch:Char;
 Begin
  Assign(f,'Input.Txt');
  Reset(f);
  While Not Eof(f) Do Begin
   Read(f,ch);
   Write(ch);
  End;
  Close(f);
 End.
```

Эта программа выдаст правильный результат. Но если вы измените Read(*f*,*ch*); Write(*ch*); на Read(*f*,*s*); Write(*s*);, где *s* имеет тип String, то программа опять зацикливается. Объясните почему.

Найдите еще варианты, при которых программа будет зацикливаться.

Измените программу еще раз:

```
Program My15_1mm;
 Var f:Text;
     ch:Char;
 Begin
  Assign(Input,'Input.Txt');
  Reset(Input);
  While Not Eof Do Begin
   Read(ch);
   Write(ch);
  End;
  Close(Input);
 End.
```

Запуск этой программы не приводит к каким-либо ошибкам и зацикливаниям. Что мы сделали? Просто переопределили стандартный ввод с клавиатуры на ввод из нашего файла на диске. Файловую переменную в этом случае можно не записывать в операторах Read и Write.

Добавьте после процедуры Close(Input) операторы ReadLn(*s*); и WriteLn(*s*);, где *s* имеет строковый тип данных. Запуск программы приводит к ошибке `Error 104: File not open for input` (файл не открыт для ввода).

Вставим после Close(Input) процедуру Reset(Input). Программа заработала. Однако вместо ввода с клавиатуры строки *s* мы вводим первую строку из файла Input.Txt. Как вернуться к вводу с клавиатуры?

Продолжим рассмотрение примеров. Запустите программу.

```
Program My15_2;
 Uses Crt;
 Const MaxN=100;
 Type MyArray=Array[1..MaxN] Of Integer;
 Var A:MyArray;
     i,j:Integer;
 Begin
  ClrScr;
  Assign(Input,'Input.Txt');
  Reset(Input);
  i:=0;
  While Not Eof Do Begin
   i:=i+1;
   Read(A[i]);
  End;
  Close(Input);
  For j:=1 To i Do Write(A[j]:2);
 End.
```

Результат — ошибка `Error 106: Invalid numeric format` (неправильный числовой формат). Мы заполняем данными из файла массив целых чисел *A*, система пытается преобразовать получаемые символы в числа, но содержимое файла — вовсе не числа. Изменим файл Input.Txt на следующий:

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

Программа работает правильно. В массиве *A* оказались числа из файла.

Изменим файл Input.Txt.

```
1
2  22
3  3  333
4  4  4  4444
5  5  5  5  55555
```

Все почти правильно. И последние записи в строках правильно преобразовались в числа, правда, кроме 55555. Вместо этого числа выводится нечто странное –9981. Если вы хорошо усвоили материал предыдущих занятий, то суть ошибки очевидна. Если нет, то вставьте перед программой {$R+} и проанализируйте ошибку.

Вернемся к посимвольному вводу из файла.

```
Program My15_2m;
 Uses Crt;
 Const MaxN=100;
 Type MyArray=Array[1..MaxN] Of Integer;
 Var A:MyArray;
    i,j,k:Integer;
    ch:Char;
 Begin
  ClrScr;
  Assign(Input,'Input.Txt');
  Reset(Input);
  i:=0;
  While Not Eof Do Begin
   Read(ch);
   {Val(ch,j,k);}
   {If k=0 Then Begin} i:=i+1;
                  A[i]:=Ord(ch)-Ord('0');
            { End;}
  End;
  Close(Input);
  For j:=1 To i Do Write(A[j]:6);
 End.
```

Результат работы программы — странная последовательность: 1 –35 –38 2 –16 2 2 –35 –38 3 ... , в которой, однако, есть определенная закономерность. Найдите ее.

Уберите фигурные скобки и вновь запустите программу. Результат: 1 2 2 2 3 3 3 3 3 ...

Измените программу так, чтобы результат все же соответствовал входному файлу 1 2 22 3 3 333 ..., но ввод осуществлялся посимвольно.

В следующем примере показано переопределение выходного файла. Запустив программу, вы обнаружите, что на экране монитора ничего нет. Это естественно, поскольку вывод произ-

водится в файл Output.Txt. Откройте этот файл. Разместите на экране все три файла — файл с текстом программы, Input.Txt, Output.Txt — так, чтобы видеть содержимое всех трех. Измените несколько раз файл Input.Txt, при этом запуская программу.

```
Program My15_3;
 Uses Crt;
 Var ch:Char;
 Begin
  ClrScr;
  Assign(Input,'Input.Txt'); Reset(Input);
  Assign(Output,'Output.Txt'); ReWrite(Output);
  While Not Eof Do Begin
   Read(ch);
   Write(ch)
  End;
  Close(Input);
  Close(Output);
 End.
```

После строки с процедурами Close(Input); Close(Output); вставьте следующий текст программы:

```
Assign(Input,'Con');
Assign(Output,'Con');
Reset(Input);
Rewrite(Output);
ReadLn(s);
WriteLn(s);
```

Когда программа остановится в состоянии ожидания ввода данных, наберите произвольный текст на клавиатуре и нажмите клавишу Enter. Этот пример показывает, как восстановить стандартный режим ввода и вывода данных.

Экспериментальный раздел занятия

1. Известно, что в файле записаны целые числа, например из диапазона от –100 до 100. Необходимо подсчитать, сколько раз каждое число встречается в файле.

Для решения задачи требуется создать файл с целыми числами. Сделаем это с помощью следующей программы.

```
Program My15_4;
 Const MaxN=10000;
       a=100; b=200; c=15;
```

```
Var i,j,k:Integer;
Begin
 Randomize;
 Assign(Output,'Number.Txt');
 ReWrite(Output);
 i:=0;
 k:=0;
 While i<MaxN Do Begin
  j:=Integer(Random(b))-a;
  i:=i+1;
  k:=k+1;
  Write(j);
  Write(' ');
  If k=c Then Begin
    WriteLn;
    k:=0;
   End;
  End;
 Close(Output);
End.
```

После работы этой программы в файле Number.Txt, созданном в текущем каталоге, содержится *MaxN* целых чисел, разбитых на строки по *c* чисел в каждой строке.

Можно ли записать в файл 100 000 чисел? Простое изменение константы *MaxN* приводит к зацикливанию. Причина простая — мы забыли изменить тип данных у переменной *i* на LongInt.

После каждого числа в файл записывается символ пробела и после *c* чисел выполняется переход на новую строку. А нельзя ли обойтись без этого?

Изменим программу, возьмем в фигурные скобки текст:

```
{Write(' ');
If k=c Then Begin WriteLn;k:=0;End;}
```

На стадии выполнения программы будут выданы два сообщения. Первое сообщение: `Error. Line too long, truncated` (строка очень длинная, она обрезается); второе сообщение, представляет собой предупреждение: `Warning. Error encountered reading file NUMBER.TXT` (ожидается ошибка при чтении файла).

Вернитесь к исходному тексту программы, а в качестве дополнительного упражнения попробуйте сформировать файл, у которого и количество чисел в строке формируется случайным образом.

Настала пора подсчитать, сколько раз каждое из чисел встречается в файле. Вторая часть задачи решается с помощью следующей программы.

```
Program My15_5;
 Uses Crt;
 Const MaxN=100;c=20;
 Type MyArray=Array[-Maxn..MaxN] Of Integer;
 Var A:MyArray;
    i,j:Integer;
 Begin
  ClrScr;
  FillChar(A,SizeOf(A),0);
     {Подсчитываем, сколько раз каждое число
     встречается в файле. Число используется как
     индекс к элементу массива A. Обратите на этот
     факт особое внимание. Это «принцип косвенной
     адресации» - одна из ключевых идей
     информатики, он реализован как на уровне
     аппаратуры компьютера, так и в любых системах
     программирования.}
  Assign(Input,'Number.Txt');
  Reset(Input);
  While Not Eof Do Begin
    Read(i);
    A[i]:=A[i]+1;
  End;
  Close(Input);
  Assign(Output,'Count.Txt'); {Записываем
         результат в файл Count.Txt, не разбивая
         на строки.}
  ReWrite(Output);
  For i:=-MaxN To MaxN Do Write(A[i],' ');
  Close(Output);
  Assign(Input,'Count.Txt'); {Открываем файл
        Count.Txt для чтения.}
  Reset(Input);
```

```
Assign(Output,'Con'); {Результат выводим
      на экран.}
ReWrite(Output);
j:=0;
While Not Eof Do Begin
 Read(i);
 j:=j+1;
 Write(i,' ');
 If j=c Then Begin
   WriteLn;
   j:=0;
  End; {Разбиваем на строки, иначе трудно
       просматривать.}
End;
End.
```

Методическое замечание

На экране должны быть обозримы все файлы: с первой и второй программами, Number.Txt и Count.Txt.

Эта программа работает вполне корректно. Но что будет, если в исходном файле окажется 10 000 одинаковых чисел? Что произойдет в этом случае?

Подсчитайте, сколько раз встречается каждый символ в соответствии с кодировкой ASCII в некотором текстовом файле.

Вот еще одна похожая задача: создается файл, содержащий не очень большое количество слов. Затем из этих слов формируется массив. Из массива случайным образом выбирается слово и записывается во второй файл, это действие выполняется 10 000 раз. После того как второй файл сформирован, необходимо решить аналогичную задачу — подсчитать, сколько раз встречается каждое слово и результат записать в третий файл.

2. Дан текстовый файл, содержащий программу на языке Паскаль. Проверить правильность расстановки скобок (круглых, квадратных, фигурных) в тексте этой программы.

Приведем простые примеры для пояснения сути задачи:

- (()) — скобки в строке расставлены правильно;
- {(] } — скобки в строке расставлены неправильно;
- ({[}]) — скобки в строке расставлены неправильно.

Между скобками допустима запись любых символов. Но если встречается какая-либо из закрывающих скобок, то последняя открывающая должна быть такого же типа. Это основной критерий проверки правильности расстановки скобок в тексте программы.

В качестве теста будущего решения будем использовать следующую простую программу:

```
Program My15_sk;
Begin
  WriteLn('{*9((()))[[[[]]]]*}');
 End.
```

Для начала выясним ASCII-коды рассматриваемых в задаче скобок. Эта задача решается так:

```
Program My15_p;
 Uses Crt;
 Begin
  ClrScr;
  WriteLn(Ord('(')); WriteLn(Ord(')'));
  {Ответ: 40, 41.}
  WriteLn(Ord('[')); WriteLn(Ord(']'));
  {Ответ: 91, 93.}
  WriteLn(Ord('{')); WriteLn(Ord('}'));
  {Ответ: 123, 125.}
  ReadLn;
 End.
```

Видим, что разность кодов для соответствующих друг другу скобок не превышает значения «два».

Хранить весь текст из файла в памяти нецелесообразно, он может быть очень большим, да это и не требуется в задаче. Необходимо хранить только открывающие скобки. Определим для этой цели массив с типом элементов Char и предположим, что уровень вложенности скобок не превышает 100.

Если очередной считанный из файла символ — открывающая скобка, то она записывается в массив. При обнаружении закрывающей скобки сравнивается ее код с кодом последней скобки, записанной в массив. Если значение разности больше двух, то фиксируется ошибка в расстановке скобок, если разность равна двум — последняя открывающая скобка удаляется из массива, и просмотр текста продолжается.

```
{$R+}
Program My15_6;
 Const MaxN=100;
 Type MyArray=Array[1..MaxN] Of Char;
 Var A:MyArray;
     ch:Char;
     yk:Integer;
     ff:Boolean;
 Begin
  FillChar(A,SizeOf(A),0);
  yk:=0;
  ff:=True;
  Assign(Input,'MY15_SK.PAS');
  Reset(Input);
   While (Not Eof) And ff Do Begin
    Read(ch);
    If (ch='(') Or (ch='[') Or (ch='{') Then Begin
        yk:=yk+1;
        A[yk]:=ch;
       End
     Else
      If (ch=')') Or (ch=']') Or (ch='}') Then
       If Ord(ch)-Ord(A[yk])<=2 Then yk:=yk-1
        Else ff:=False;
   End;
  Close(Input);
  If ff Then WriteLn('Скобки расставлены
                      правильно')
   Else WriteLn('В тексте программы есть ошибка
                 в расстановке скобок');
End.
```

Откроем в различных окнах тексты двух программ — последней и My15_sk и приступим к экспериментам. Вносим изменение во вторую программу, сохраняя ее текст, переходим в другое окно, запускаем программу и анализируем результат.

Например, при некоторых исходных данных возникает ошибка `Error 201: Range check error`. Программа содержит логическую неточность. Устраните ее.

Измените программу так, чтобы, например, проверялась правильность записи слова WriteLn в тексте программы.

Придумайте еще ряд модификаций программы.

3. В программе, записанной на диск под именем My15_7.pas, подсчитывается количество строк в текстовом файле. При этом каждая строка сохраняется как элемент массива для последующего анализа и строки выводятся на экран. Получен один из вариантов программы вывода собственного текста на экран, причем не лучший. Попробуйте предложить как можно больше способов решения этой проблемы, разумеется, с их реализациями.

```
Program My15_7;
 Const MaxN=100;
 Type MyArray=Array[1..MaxN] Of String;
 Var A:MyArray;
    cnt,i:Integer;
 Begin
  cnt:=0;
  Assign(Input,'MY15_7.PAS'); Reset(Input);
  While Not Eof  Do Begin
   cnt:=cnt+1;
   ReadLn(A[cnt]);
  End;
  Close(Input);
  WriteLn(cnt);
  For i:=1 To cnt Do WriteLn(A[i]);
 End.
```

Программы, выводящие на экран свой собственный текст, без использования хранящегося на диске файла с собственным текстом, называют интроспективными. Таким образом, наша программа не является интроспективной. Эта головоломка была одно время довольно популярна среди программистов. Попробуйте свои силы в решении данной проблемы.

Задания для самостоятельной работы

1. Дан текстовый файл, содержащий целые числа. Найдите:

- количество чисел в файле;
- максимальное число в файле, в каждой строке файла;
- сумму чисел в файле, в каждой строке;
- разность между максимальным и минимальным числами для каждой строки и для всего файла;
- среднее арифметическое чисел в файле, в каждой строке;

- номер максимального числа в файле;
- сумму максимальных чисел в файле;
- сумму четных чисел в файле.

2. Дан текстовый файл, содержащий строки. Найдите:

- количество строк, начинающихся с заглавных латинских букв;
- количество строк, начинающихся и заканчивающихся одинаковыми символами;
- самые короткие строки;
- симметричные строки (палиндромы).

3. Дан текстовый файл, содержащий строки. Вставьте в начало каждой строки ее номер и запишите преобразованные строки в новый файл.

4. Даны два текстовых файла. Запишите в третий файл только те строки, которые есть и в первом, и во втором файлах.

5. Дан текстовый файл. Допишите в его конце следующие данные: количество строк, количество символов в каждой строке, количество чисел в каждой строке.

6. Даны два файла *A* и *B* (тип элементов одинаковый). Поменяйте местами содержимое этих файлов. Использовать процедуру Rename не разрешается.

7. Содержимое текстового файла копируется в другой файл, при этом каждая строка циклически сдвигается вправо на n символов. Пример циклического сдвига строки abcdefqwrt на 3 символа: wrtabcdefq. Решите ту же задачу, сдвигая каждое слово на половину его длины.

8. Дано некоторое множество слов. Исключите их из текстового файла. Например, из файла с текстом программы исключите все слова Begin и End.

9. Считаем, что длина строк текстового файла не превышает 80 символов. Преобразуйте файл так, чтобы все строки были отцентрированы: более короткие строки дополняются символами пробела, текст строки размещается по ее центру.

10. Дан файл, в котором встречаются даты. Каждая дата — это число, месяц и год (например, 13.05.1949 г.). Найдите самую раннюю дату.

1.3. Массив – фундаментальная структура данных

Занятие № 16. Методы работы с элементами одномерного массива

План занятия

1. Примеры коротких программ для обработки элементов массива.
2. Эксперименты с программами (вставка и удаление элементов массива).
3. Выполнение самостоятельной работы.

Примеры коротких программ

В этих примерах используется следующее описание данных:

```
Const MaxN=…; {Максимальное количество элементов
                в массиве.}
Type MyArray=Array[1..MaxN] Of Integer;
```

Основная программа может иметь следующий вид:

```
Begin
 Init(m,X); {Ввод элементов массива и
          инициализация глобальных переменных.}
 SolveProc(<параметры>); {Вызов основной
          процедуры решения задачи.}
 Print(m,X); {Вывод результата.}
 ReadLn;
End.
```

Если для решения задачи необходимо использовать функцию, то возможна такая организация программы:

```
Begin
 Init(m,X); {Ввод элементов массива
        и инициализация глобальных переменных.}
 WriteLn(SolveFun(<параметры>)); {Вызов функции,
    если результат ее действий выводится спомощью
    одного оператора WriteLn.}
 ReadLn;
End.
```

1. Заменить отрицательные элементы массива на противоположные по знаку.

```
Procedure NegPos(n:Integer;Var A: MyArray);
 Var i: Integer;
 Begin
  For i:=1 To n Do
   If A[i]<0 Then  A[i]:=-A[i];
 End;
```

2. Прибавить к каждому элементу массива число 25.

```
Procedure Add25( n:Integer;Var A:MyArray);
 Var i: Integer;
 Begin
  For i:=1 To n Do A[i]:=A[i]+25;
 End;
```

3. Прибавить к четным элементам массива первый элемент, а к нечетным — последний. Первый и последний элементы не изменять.

```
Procedure AddFirstLast(n:Integer;Var A:MyArray);
 Var i: Integer;
 Begin
  For i:=2 To n-1 Do
   If A[i] Mod 2=0 Then A[i]:=A[i]+A[1]
    Else A[i]:=A[i]+A[n];
 End;
```

4а. Даны два одномерных массива одинаковой размерности. Получить третий массив такой же размерности, каждый элемент которого равен сумме соответствующих элементов данных массивов.

```
Procedure AddArray(n:Integer;A,B:MyArray;
                           Var C: MyArray );
 Var i: Integer;
 Begin
  For i:=1 To n Do C[i]:=A[i]+B[i];
 End;
```

4б. Даны два целочисленных массива, состоящие из одинакового количества элементов. Получить третий массив той же

размерности, каждый элемент которого равен наибольшему из соответствующих элементов данного массива.

```
Procedure AddMaxArray(n:Integer;A,B:MyArray;
                              Var C: MyArray );
 Var i: Integer;
 Begin
  For i:=1 To n Do
   If A[i]>B[i] Then C[i]:=A[i] Else C[i]:=B[i];
 End;
```

5. Дан первый элемент арифметической прогрессии и разность между соседними элементами. Сформировать одномерный массив из первых n элементов арифметической прогрессии.

Пусть *one* — первый элемент прогрессии, а k — разность. Если известно значение элемента с номером $i-1$, то элемент с номером i вычисляется по правилу: $a_i=a_{i-1}+k$, или $a_i=one+k\cdot(i-1)$.

```
Procedure FormProg(one, k, n: Integer;
                       Var A: MyArray);
 Var i: Integer;
 Begin
  A[1]:=one;
  For i:=2 To n Do A[i]:=A[i-1]+k;
 End;
```

Для геометрической прогрессии очередной элемент вычисляется по правилу $a_i=a_{i-1}\cdot k$, где k — знаменатель прогрессии. Найти сумму первых n элементов прогрессии.

```
Function FormProgm(one, k, n: Integer):LongInt;
 Var i,t: Integer;
     sum:LongInt;
 Begin
  sum:=one;
  t:=one;
  For i:=2 To n Do Begin
    t:=t*k;
    sum:=sum+t;
   End;
  FormProgm:=sum;
 End;
```

6. Даны два одномерных массива A и B. Найти их скалярное произведение. Скалярным произведением двух массивов одинаковой размерности назовем сумму произведений соответствующих элементов:

$A[1] \cdot B[1]+A[2] \cdot B[2] +\ldots+A[n-1] \cdot B[n-1]+A[n] \cdot B[n]$,

где n — количество элементов в массивах.

```
Function Product(n:Integer;A,B:MyArray):LongInt;
 Var i: Integer;
     s: LongInt;
 Begin
  s:=0;
  For i:=1 To n Do s:=s+A[i]*B[i];
  Product:=s;
 End;
```

7а. Найти максимальный по модулю элемент массива.

```
Function MaxAbs(n:Integer; A:MyArray):Integer;
 Var i,max:Integer;
 Begin
  max:=Abs(A[1]);
  For i:=2 To n Do
   If Abs(A[i])>max Then max:=Abs(A[i]);
  MaxAbs:=max;
 End;
```

7б. Найти целую часть среднего арифметического значения элементов массива.

```
Function Average(n:Integer; A:MyArray):Integer;
 Var i,sum:Integer;
 Begin
  sum:=A[1];
  For i:=2 To n Do sum:=sum+A[i];
  Average:=sum Div n;
 End;
```

7в. Изменить знак у максимального по модулю элемента массива.

```
Procedure SignMaxElem(n:Integer; Var A:MyArray);
 Var i,j,Max:Integer;
```

```
Begin
 Max:=Abs(A[1]);
 j:=1;
 For i:=2 To n Do
  If Abs(A[i])>Max Then Begin
    Max:=Abs(A[i]);
   j:=i;
  End;
 A[j]:=-A[j];
End;
```

8. Все четные элементы массива возвести в квадрат, а нечетные удвоить.

```
Procedure MaxEven(n:Integer; Var A:MyArray);
 Var i:Integer;
 Begin
   For i:=1 To n Do
    If A[i] Mod 2 =0 Then A[i]:=A[i]*A[i]
    Else A[i]:=2*A[i];
 End;
```

9. Из положительных элементов массива вычесть элемент с номером $k1$, отрицательные увеличить на значение элемента с номером $k2$, а элементы, равные нулю, оставить без изменения.

```
Procedure ChangeElem(n,k1,k2:Integer;
                     Var A:MyArray);
 Var i,w,v:Integer;
 Begin
  w:=A[k1];
  v:=A[k2];
  For i:=1 To n Do
   If A[i] >0 Then A[i]:=A[i]+w
    Else If A[i]<0 Then A[i]:=A[i]-v;
 End;
```

Операторы w:=A[$k1$]; v:=A[$k2$]; здесь обязательны. Соответствующий фрагмент нельзя переписать в виде:

```
If A[i] >0 Then A[i]:=A[i]+A[k1]
 Else If A[i]<0 Then A[i]:=A[i]-A[k2];
```

Проверьте и объясните.

10. Из элементов массива A сформировать элементы массива B по правилу: $B[i]:=A[1]+A[2]+...+A[i]$.

```
Procedure SumElem(n:Integer; A:MyArray;
                  Var B:MyArray);
 Var i:Integer;
 Begin
  B[1]:=A[1];
  For i:=2 To n Do B[i]:=B[i-1]+A[i];
 End;
```

Экспериментальный раздел занятия

1. *Удаление из массива* последнего (в том случае, если их несколько) максимального элемента. Для решения задачи необходимо:

- найти номер k последнего максимального элемента;
- сдвинуть все элементы, начиная с $(k+1)$-го, на один элемент влево;
- последнему элементу присвоить значение 0.

```
{$R+}
Program My16_1;
 Const MaxN=10;la=-10;ha=21; {Размер массива
                              и интервал значений.}
 Type MyArray=Array[1..MaxN] Of Integer;
 Var A:MyArray;
     k:Integer;
 Procedure Init(t,v,w:Integer;Var X:MyArray);
        {Напомним процедуру формирования значений
        элементов массива.}
  Var i:Integer;
  Begin
   Randomize;
   For i:=1 To t Do X[i]:=v+Integer(Random(w));
  End;
 Procedure Print(t:Integer;X:MyArray); {Вывод t
        первых элементов массива X.}
  Var i:Integer;
  Begin
   For i:=1 To t Do Write(X[i]:5);
   WriteLn;
  End;
```

```
Function Max(t:Integer;X:MyArray):Integer;
{Поиск номера последнего максимального элемента.}
 Var i,Mx:Integer;
 Begin
  Mx:=1;
  For i:=2 To t Do
   If X[i]>X[Mx] Then Mx:=i;
  Max:=Mx;
 End;
Procedure Sz(t,q:Integer;Var X:MyArray); {Сдвиг
          элементов массива влево на одну позицию,
          начиная с элемента с номером q+1.}
 Var i:Integer;
 Begin
  For i:=q To t-1 Do X[i]:=X[i+1];
  X[t]:=0;
 End;
Begin
 n:=MaxN;
 Init(n,la,ha,A);
 WriteLn('Исходный массив.');
 Print(n,A);
 k:=Max(n,A);
 Sz(n,k,A);
 WriteLn('Массив после удаления элемента.');
 Print(n-1,A);
 ReadLn;
End.
```

Предположим, что максимальный элемент встречается несколько раз. Во-первых, следует изменить функцию Max. Требуется находить не номер максимального элемента, а его значение.

```
Function Max(t:Integer;X:MyArray):Integer;
 Var i,Mx:Integer;
 Begin
  Mx:=X[1];
  For i:=2 To t Do
   If X[i]>Mx Then Mx:=X[i];
  Max:=Mx;
 End;
```

Затем повторно просматриваем массив, удаляя элементы, равные максимальному, и уменьшая количество элементов в массиве.

```
Procedure Solve(Var t:Integer;Var X:MyArray);
 Var k,i:Integer;
 Begin
  k:=Max(t,X);
  i:=t+1;
  While i>1 Do Begin
    i:=i-1;
    If X[i]=k Then Begin
     Sz(t,i,X);
     t:=t-1;
    End;
  End;
End;
```

Измените программу так, чтобы при удалении массив просматривался не с конца, а с начала.

2. *Вставка элемента в массив*. Необходимо вставить элемент после первого максимального элемента массива (пусть его номер равен q). Операция выполняется следующим образом:

- первые q элементов массива остаются без изменений;
- все элементы, начиная с $(q+1)$-го, необходимо сдвинуть на одну позицию вправо;
- на место $(q+1)$ записываем значение константы *InsC*.

```
{$R+}
Program My16_2;
 Const MaxN=10;la=1;ha=6;InsC=100;
 Type MyArray=Array[1..MaxN+1] Of Integer;
{Вставляем один элемент. Резервируем для него место.}
 Var A:MyArray;
     n,k:Integer;
 Procedure Init(t,v,w:Integer;Var X:MyArray);
{Формируем исходный массив.}
Procedure Print(t:Integer;X:MyArray); {Вывод
                    значений элементов массива.}
Function Max(t:Integer;X:MyArray):Integer; {Поиск
          номера первого максимального элемента.}
```

```
 Var i,Mx:Integer;
 Begin
  Mx:=1;
  For i:=2 To t Do
   If X[i]>X[Mx] Then Mx:=i;
  Max:=Mx;
 End;
Procedure InsPs(t,q:Integer;Var X:MyArray);
  {Сдвиг на одну позицию вправо и вставка на место
   q+1 (после заданного элемента) элемента,
   равного InsC.}
 Var i:Integer;
 Begin
  For i:=t DownTo q+1 Do X[i+1]:=X[i];
  X[q+1]:=InsC;
 End;
Begin
 n:=MaxN;
 Init(n,la,ha,A);
 WriteLn('Вывод массива до вставки элемента.');
 Print(n,A);
 k:=Max(n,A); {Определяем место первого
               максимального элемента.}
 InsPs(n,k,A);
 WriteLn('Вывод массива после вставки
         элемента.');
 Print(n+1,A);
 ReadLn;
End.
```

Для вставки элемента перед заданным элементом требуется незначительное изменение процедуры InsPs.

```
Procedure InsPs(t,q:Integer;Var X:MyArray);
 Var i:Integer;
 Begin
  For i:=t DownTo q Do X[i+1]:=X[i];
  X[q]:=InsC;
 End;
```

Изменим задачу — пусть требуется вставить новые элементы, равные *InsC*, после всех максимальных элементов массива.

Во-первых, следует изменить размер массива. Если массив состоит из одинаковых элементов, то в результате наших действий он увеличится в два раза. Поэтому изменим описание массива:

```
Type MyArray=Array[1..2*MaxN] Of Integer;
```

Во-вторых, предыдущее решение в принципе не изменяется. Функция Max(n,A) и процедура InsPs(n,k,A) включаются в новую процедуру Solve.

```
Procedure Solve(Var t:Integer;Var X:MyArray);
 Var k,i:Integer;
 Begin
  k:=Max(t,X); {Определяем значение максимального
                элемента.}
  i:=1;
  While i<=t Do Begin
    If X[i]=k Then Begin
      InsPs(t,i,X); {Вставляем элемент
                 и увеличиваем размер массива.}
      t:=t+1;
     End;
    i:=i+1;
  End;
 End;
```

При всех ли исходных данных эта программа работает правильно? Оказывается, нет. Не трогая программный код, а изменяя только значения констант в программе, найдите ошибку. Эксперименты с программой приведут вас к знакомой ошибке: `Error 202: Range check error`, а она уже «натолкнет» на неточность в программе.

Измените программу так, чтобы вставка выполнялась перед всеми максимальными элементами массива. После этого необходимо добиться, чтобы вставка выполнялась и перед максимальными, и после максимальных элементов.

Задания для самостоятельной работы

1. Замените:
   - элементы с k_1-го по k_2-й на противоположные по знаку;
   - первый отрицательный элемент нулем;
   - максимальный элемент на противоположный по знаку;

- первый элемент, кратный 5, нулем;
- элементы с нечетными номерами на квадраты их номеров;
- последний положительный элемент на второй элемент массива;
- все элементы между минимальным и максимальным элементами массива нулями;
- все элементы, кратные 3, на третий элемент массива;
- все элементы с нечетными номерами разделить нацело на первый элемент.

2. Из элементов массива A сформируйте массив B той же размерности следующим образом:
   - первые 10 элементов — по правилу $B[i]:=A[i]+i$, остальные — по правилу $B[i]:=A[i]-i$;
   - если номер четный, то $B[i]:=i \cdot A[i]$, если нечетный, то $B[i]:=-A[i]$;
   - элементы с 3-го по 12-й — по правилу $B[i]:=-A[i] \cdot A[i]$, все остальные — по правилу $B[i]:=A[i]-1$;
   - если номер четный, то $B[i]:=A[i] \cdot A[i]$, если нечетный, то $B[i]:=A[i]$ Div i.
3. Сформируйте массив A с помощью датчика случайных чисел целыми числами из следующих интервалов: [–21, 53], [–43, 32], [–11, 67], [–35,75], [–45,95].
4. Удалите из массива все элементы:
   - содержащие в своей записи цифру 5;
   - состоящие из одинаковых цифр (включая однозначные числа);
   - последняя цифра которых четная, а само число делится на нее;
   - первая цифра которых четная;
   - кратные 7 и принадлежащие промежутку [a, b] (a и b вводятся с клавиатуры);
   - значения которых меньше нуля;
   - большие данного числа a (a вводится с клавиатуры);
   - четные, стоящие на нечетных местах;
   - которые повторяются (т. е. оставив только их первые вхождения, получить массив из различных элементов);
   - кратные 3 или 5;
   - начиная с $k1$-го по $k2$-й ($k1$ и $k2$ вводятся с клавиатуры).

5. Вставьте число k (все числа, указанные в формулировках задач, вводятся с клавиатуры):
   - после каждого элемента, кратного своему номеру;
   - перед каждым элементом, в котором есть цифра один;
   - перед и после всех элементов, заканчивающихся на данную цифру;
   - после всех элементов, больших заданного числа, и число t — перед всеми элементами, кратными 3;
   - между всеми соседними элементами, которые образуют пару с одинаковыми знаками;
   - после первого отрицательного элемента;
   - перед последним отрицательным элементом;
   - после максимального элемента, и число t – перед максимальным элементом;
   - перед всеми элементами, кратными заданному числу;
   - перед всеми отрицательными элементами;
   - после всех элементов, больших данного числа p, и число t — перед всеми элементами, меньшими числа p;
   - перед всеми элементами, большими k, и число t — после всех элементов, меньших его.
6. Переставьте в массиве:
   - первые три и последние три элемента местами, не нарушая в каждой тройке порядок следования;
   - первый положительный и последний отрицательный элементы;
   - первый элемент и максимальный;
   - второй и минимальный;
   - первый и последний отрицательный;
   - элементы следующим образом: $A[1]$, $A[12]$, $A[2]$, $A[11]$, …, $A[5]$, $A[8]$, $A[6]$, $A[7]$;
   - первые k элементов в конец, т. е.: $A[k+1]$, $A[k+2]$, …, $A[n]$, $A[1]$, $A[2]$, …, $A[k]$;
   - в обратном порядке часть массива между элементами с номерами k_1 и k_2, включая их.
   - первый элемент с последним, второй с предпоследним и т. д.;
   - в обратном порядке элементы, расположенные между минимальным и максимальным элементами;
   - элементы по следующим правилам ($2n$ элементов в массиве):

$A[n+1]$, $A[n+2]$,..., $A[2n]$, $A[1]$, $A[2]$,..., $A[n]$;
$A[n+1]$, $A[n+2]$,..., $A[2n]$, $A[n]$, $A[n-1]$,..., $A[1]$;
$A[1]$, $A[n+1]$, $A[2]$, $A[n+2]$,..., $A[n]$, $A[2n]$;
$A[2n]$, $A[2n-1]$,..., $A[n+1]$, $A[1]$, A[2],..., A[n].

7. Дан одномерный массив целых чисел. Найдите количество различных чисел среди элементов массива.
8. Дан одномерный массив целых чисел, значения которых находятся в интервале от a до b. Найдите количество различных чисел среди элементов массива.
9. Дан одномерный массив целых чисел и число q. Переставьте элементы массива так, чтобы сначала были записаны все числа, меньшие q, а далее — числа, большие или равные q. Дополнительные массивы использовать не разрешается. Модифицируйте решение для случая: меньшие q, затем равные q и далее — большие q.
10. Дан массив из n целых чисел и число m ($m \le n$). Для каждых m последовательных элементов массива вычислите их сумму (количество таких последовательностей равно $n-m+1$).

Занятие № 17. Двумерные массивы. Работа с элементами

План занятия

1. Структура двумерного массива и его описание.
2. Шаблон для решения задач с двумерными массивами.
3. Эксперименты с программами (поиск максимального элемента в массиве; формирование значений элементов одномерного массива; поиск элементов с определенными свойствами; заполнение массива по заданным правилам).
4. Выполнение самостоятельной работы.

Структура двумерного массива и его описание

Массивы, положение элементов в которых описывается двумя индексами, называют двумерными. Логическая *структура* такого массива может быть представлена прямоугольной матрицей. Каждый элемент матрицы однозначно определяется указанием номера строки и номера столбца. Но память компьютера — это аналог одномерного массива. Как происходит отображение логической структуры двумерного массива в фи-

зическую? Известны два способа его размещения: построчное и по столбцам. В конкретной системе программирования используется один из них.

Рассмотрим первый, как обычно, на примере. Пусть дан двумерный массив A из целых чисел (тип Integer — 2 байта). Количество его строк равно 5, количество столбцов — 4. Пусть первый элемент $A[1,1]$ записан в ячейке с номером 1000. Вычислим адрес $A[4,3]$.

$\text{Addr}(A[4,3])=1000+2\cdot(4\cdot(4-1)+(3-1))=1028$.

В каждой строке записано по 4 элемента. В трех строках — 12 элементов, в 4-й строке до 3-го элемента записано 2 элемента. Сложив, умножаем полученное число элементов на 2, поскольку каждый элемент занимает 2 байта.

В общем случае для массива $A[n, m]$ из элементов, занимающих v байт памяти каждый, формула имеет вид:

$\text{Addr}(A[i, j])=\text{Addr}(A[1,1])+v\cdot(m\cdot(i-1)+(j-1))$.

Описать двумерный массив на языке Паскаль можно по-разному.

```
Const MaxN=…; MaxM=…; {Максимальные значения
количества строк и столбцов двумерного массива.}
```

Первый способ:

```
Type OMyArray=Array[1..MaxM] Of Integer;
{Одномерный массив из целых чисел.}
TMyArray=Array[1..MaxN] Of OMyArray; {Одномерный
массив, элементами которого являются одномерные
массивы из целых чисел.}
```

Второй способ:

```
Type TMyArray=Array[1..MaxN,1..MaxM] Of Integer;
{Двумерный массив из целых чисел.}
```

Будем отдавать предпочтение второму способу описания двумерного массива.

Шаблон для решения задач с двумерными массивами

Для работы с двумерными массивами изготовим *программу-шаблон*. Она включает ввод данных из файла и вывод элементов двумерного массива в файл. Размеры двумерного массива (n — число строк и m — число столбцов) вводятся из файла. Далее по тексту, если этого не требует задача, процедуры ввода и вывода двумерного массива не приводятся.

```
{$R+}
Program My17_1;
 Const MaxN=10; MaxM=8;
 Type TMyArray=Array[1..MaxN,1..MaxM] Of Integer;
 Var A:TMyArray;
     n,m:Integer;
 Procedure TInit(Var v,w:Integer;Var X:TMyArray);
  Var i,j:Integer;
  Begin
   Assign(Input,'Input.Txt');
   Reset(Input);
   ReadLn(v,w); {В первой строке файла записаны
                 значения n и m.}
   For i:=1 To v Do
    For j:=1 To w Do Read(X[i,j]);
   Close(Input);
  End;
 Procedure TPrint(v,w:Integer;X:TMyArray);
  Var i,j:Integer;
  Begin
   Assign(Output,'Output.Txt');
   ReWrite(Output);
   For i:=1 To v Do Begin
    For j:=1 To w Do Write(X[i,j]:4);
    WriteLn;
   End;
   Close(Output);
  End;
 Begin
  TInit(n,m,A);
  TPrint(n,m,A);
 End.
```

Массив просматривается так, как это показано на рис. 1.13: сначала элементы первой строки, затем — второй и так до *n*-й строки.

Напомним, что для удобства работы экран должен быть разбит на три окна, которые обозримы одновременно. Изменение значений во входном файле (не забывайте его записывать) и запуск программы приводят к изменению выходного файла. Эти изменения видны без дополнительных операций по пере-

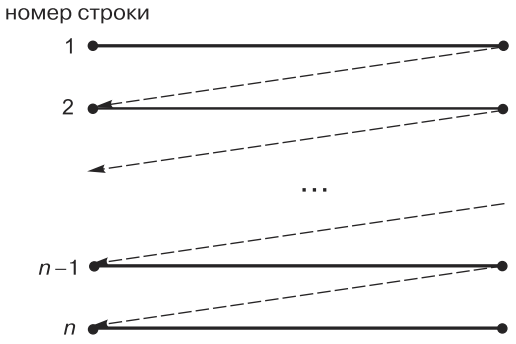


Рис. 1.13. Порядок построчного просмотра элементов массива

ходу от одного окна к другому. Текущим должен быть каталог с файлами программы, Input.Txt и Output.Txt. Названия входного и выходного файлов, естественно, могут быть другими.

Возможно формирование массива с использованием датчика случайных чисел и путем ввода данных с клавиатуры, но мы не будем останавливаться на этом.

Экспериментальный раздел занятия

1. Найти значение первого максимального элемента двумерного массива и его индексы. Результаты вывести в файл Output.Txt.

```
Procedure Search(v,w:Integer;X:TMyArray;
                 Var smax,simax,sjmax:Integer);
 Var i,j:Integer;
 Begin
  smax:=X[1,1];
  simax:=1;
  sjmax:=1;
  For i:=1 To v Do
   For j:=1 To w Do
    If X[i,j]>smax Then  Begin
      smax:=X[i,j];
      simax:=i;
      sjmax:=j;
     End;
  End;
```

Если сама процедура Search вряд ли вызывает вопросы, то вот для вывода результата в файл Output.Txt наших знаний не

хватает. Мы можем переопределить вывод значения максимального элемента с его индексами на экран монитора.

Это делается при помощи операторов:

```
Assign(Output,'Con');
Rewrite(Output);
```

'Con' означает, что в качестве устройства вывода выбран монитор.

Однако если бы требовалось вывести исходный массив в файл Output.Txt, то получилось бы, что массив выводится в файл, а результаты работы — на экран.

Воспользуемся стандартной процедурой Append. Добавленная в основную программу процедура Append(Var *f*:Text) открывает существующий файл для добавления в него информации. Переменная *f* является файловой переменной типа Text, которая должна быть связана с внешним файлом с помощью процедуры Assign. Процедура Append открывает существующий внешний файл, имя которого поставлено в соответствие переменной *f*. Если файл уже открыт, то сначала он закрывается, а затем открывается повторно. Указатель текущей позиции (аналог головки чтения-записи в магнитофоне) при этом устанавливается на конец данного файла, то есть очередная запись в файл окажется самой последней.

Добавим в основную программу фрагмент:

```
Search(n,m,A,Max,iMax,jMax);
Append(Output);
WriteLn(Max:5,iMax:5,jMax:5);
```

В конец входного файла Output.Txt будут дописаны результаты работы программы.

Измените процедуру Search для поиска последнего максимального элемента, для поиска всех максимальных элементов и их индексов.

2. Сформировать одномерный массив, каждый элемент которого равен сумме отрицательных элементов соответствующей строки заданной целочисленной матрицы.

Задача требует введения одномерного массива, поэтому приведем описание данных, процедуры подсчета суммы отрицательных элементов в строке и добавления данных в выходной файл, а также основную программу.

```
{$R+}
Program My17_2;
 Const MaxN=10; MaxM=8;
 Type TMyArray=Array[1..MaxN,1..MaxM] Of Integer;
      OMyArray=Array[1..MaxN] Of Integer;
 Var  A:TMyArray;
      Sum:OMyArray;
      n,m:Integer;
Procedure SumNeg(v,w:Integer;X:TMyArray;
                 Var Ssum:OMyArray);
 Var i,j:Integer;
 Begin
  For i:=1 To v Do Begin
    Ssum[i]:=0;
    For j:=1 To w Do
     If X[i,j]<0 Then Ssum[i]:=Ssum[i]+X[i,j];
   End;
 End;
Procedure OPrint(v:Integer;OSum:OMyArray);
 Var i:Integer;
 Begin
  Append(Output);
  For i:=1 To v Do Write(OSum[i]:4);
  WriteLn;
 End;
Begin {Основная программа.}
  TInit(n,m,A);
  TPrint(n,m,A);
  SumNeg(n,m,A,Sum);
  OPrint(n,Sum);
End.
```

3. Рассмотрим функцию SNeg, которая определяет, есть ли в массиве элемент, равный нулю:

```
Function SNeg(v,w:Integer;X:TMyArray):Boolean;
 Var i,j:Integer;
     pp:Boolean;
 Begin
  pp:=False;
  For i:=1 To v Do
```

```
  For j:=1 To w Do
   If X[i,j]=0 Then pp:=True;
  SNeg:=pp;
End;
```

Фрагмент основной программы имеет вид:

```
…
Append(Output);
If SNeg(n,m,A) Then WriteLn('Yes')
 Else WriteLn('No');
…
```

Очевидно, что у функции SNeg есть серьезный недостаток. А именно: даже если нулю равен первый элемент массива *X*[1,1], просмотр массива продолжается до конца, несмотря на его бессмысленность. Как известно, «задача определяет тип используемых конструкций повторения», поэтому используйте операторы While или Repeat – Until в функции Sneg и измените ее, устранив ненужный просмотр массива.

4. Рассмотрим функцию Sim, позволяющую определить, является ли данный квадратный массив ($n \times n$) симметричным относительно своей главной диагонали (рис. 1.14). Элементы на главной диагонали характеризуются совпадением значением индексов — $i=j$, на побочной — соотношением $i=n-j+1$, где n — размерность массива.

```
Function Sim(v,w:Integer;X:TMyArray):Boolean;
 Var i,j:Integer;
     pp:Boolean;
 Begin
  pp:=True;
  For i:=1 To v-1 Do
   For j:=i+1 To w Do
    If X[i,j]<>X[j,i] Then pp:=False;
  Sim:=pp;
 End;
```

Измените функцию Sim так, чтобы симметричность массива проверялась относительно побочной диагонали (рис. 1.14).

В предыдущем примере был рассмотрен недостаток функции SNeg. Он присущ и этой функции. Устраните его.

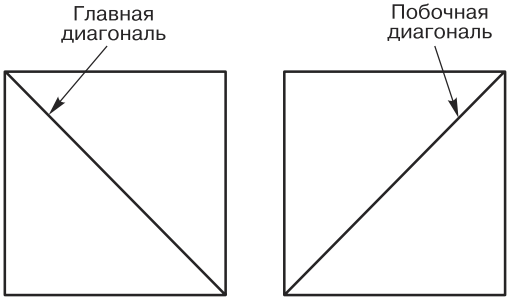


Рис. 1.14. Диагонали квадратной матрицы

5. В массиве A размерностью $n \times m$ к элементам четных столбцов прибавьте элементы первого столбца соответствующих строк.

```
Procedure Change( v,w:Integer;Var X:TMyArray);
 Var i, j: Integer;
 Begin
  For i:=1 To v Do
   For j:=1 To w Div 2 Do
       X[i,2*j]:=X[i,2*j]+X[i,1];
 End;
```

Особенностью решения задач этого типа является возможность изменения элемента, который должен использоваться в обработке. Например, прибавляем элементы первого столбца к элементам нечетных столбцов. Модификация $X[i,2 \cdot j-1]$ приведет к тому, что элементы 3-го, 5-го и так далее столбцов увеличатся на удвоенное значение элементов первого столбца. Проведите корректное изменение процедуры Change.

6. Заполнить массив A размером $n \times m$, например, для $n=6$ и $m=8$, так, как показано на рис. 1.15, т. е. в виде «змейки».

Сформулируем правило заполнения. Если номер строки — нечетное число, то $A[i,j]=(i-1) \cdot m+j$, иначе $A[i,j]=i \cdot m-j+1$.

```
Procedure Fill(v,w:Integer;Var X: TMyArray);
 Var i, j: Integer;
 Begin
  For i:=1 To v Do
   For j:=1 To w Do
    If i Mod 2=1 Then X[i,j]:=(i-1)*w+j
     Else X[i,j]:=i*w-j+1;
 End;
```

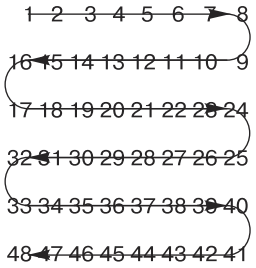


Рис. 1.15. Порядок заполнения массива

Сформируйте массивы A по следующим правилам:

1) $A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 15 & 16 & 17 & 18 & 19 & 20 & 21 \\ 22 & 23 & 24 & 25 & 26 & 27 & 28 \end{pmatrix}$;

2) $A = \begin{pmatrix} 1 & 0 & 2 & 0 & 3 & 0 & 4 \\ 0 & 5 & 0 & 6 & 0 & 7 & 0 \\ 8 & 0 & 9 & 0 & 10 & 0 & 11 \\ 0 & 12 & 0 & 13 & 0 & 14 & 0 \end{pmatrix}$;

3) $A = \begin{pmatrix} 1 & 3 & 4 & 10 & 11 & 21 \\ 2 & 5 & 9 & 12 & 20 & 22 \\ 6 & 8 & 13 & 19 & 23 & 30 \\ 7 & 14 & 18 & 24 & 29 & 31 \\ 15 & 17 & 25 & 28 & 32 & 35 \\ 16 & 26 & 27 & 33 & 34 & 36 \end{pmatrix}$;

4) $A = \begin{pmatrix} 1 & 12 & 13 & 24 & 25 & 36 \\ 2 & 11 & 14 & 23 & 26 & 35 \\ 3 & 10 & 15 & 22 & 27 & 34 \\ 4 & 9 & 16 & 21 & 28 & 33 \\ 5 & 8 & 17 & 20 & 29 & 32 \\ 6 & 7 & 18 & 19 & 30 & 31 \end{pmatrix}$.

7. Составьте программу, запрашивающую координаты ферзя на шахматной доске и показывающую поля доски, находящиеся под боем.

Заметим, что шахматную доску удобно представить в виде двумерного массива размером 8×8. Координаты ферзя определяются двумя числами (номер строки и номер столбца), но в шахматах принято вводить букву и число. Буква отвечает за номер строки, а число — за номер столбца. Поэтому не будем отступать от традиций и введем координаты именно таким образом. В программе сделаем проверку корректности ввода, и если все правильно, то переведем букву в соответствующее ей число (a — 1, b — 2, c — 3, d — 4, e — 5, f — 6, g — 7, h — 8). Для решения задачи используем ряд свойств шахматной доски. Все диагонали доски делятся на восходящие и нисходящие (рис. 1.16).

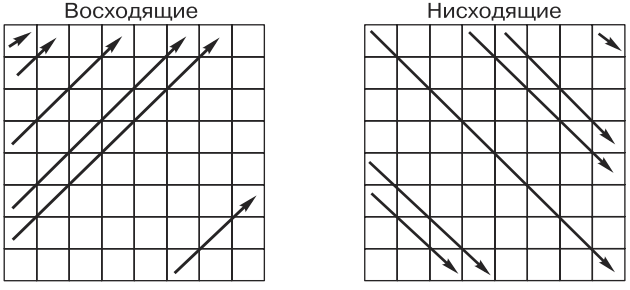


Рис. 1.16. Типы диагоналей

При этом на каждой диагонали выполняется свойство:

- для клеток любой восходящей диагонали сумма номера строки и номера столбца постоянна ($i+j$=Const);
- для клеток любой нисходящей диагонали разность номера строки и номера столбца постоянна ($i-j$=Const).

Проверьте, что для восходящих диагоналей сумма индексов изменяется от 2 до 16, а для нисходящих разность изменяется от –7 до 7.

```
Program My17_3;
 Const MaxN=8;MaxM=8;
 Type TMyArray=Array[1..MaxN,1..MaxM] Of Integer;
 Procedure TInit(v,k, l: Integer;
                 Var X: TMyArray);
```

```
 Var i, j: Integer;
 Begin
  For i:=1 To v Do
   For j:=1 To v Do
    If (i=k) Or (j=l) Or (i+j=k+l) Or (i-j=k-l)
     Then X[i,j]:=1
     Else X[i,j]:=0; {1 - клетка под боем, 0 - нет.}
  X[k,l]:=2; {В этой клетке находится ферзь.}
 End;
Procedure Solve;
 Var  A: TMyArray;
      c: Char;
      str, stl: Integer;
 Begin
  Assign(Input,'Input.Txt'); Reset(Input);
  Assign(Output,'Output.Txt'); ReWrite(Output);
  ReadLn(c, stl); {Координаты ферзя записаны
                   в файле Input.Txt.}
 If (c<'a') Or (c>'h') Or (stl<1) Or (stl>MaxN)
  Then WriteLn ('Некорректный ввод.')
  Else Begin
    str:=Ord(ch)-Ord('a')+1; {Преобразуем символ
                              в номер строки.}
    TInit(MaxN,str,stl,A);
    TPrint(MaxN,MaxM,A); {Процедура совпадает
                          с ранее рассмотренной.}
   End;
  Close(Input);
  Close(Output);
End;
```

Основная программа состоит из одной строки — вызов процедуры Solve. Измените решение так, чтобы фиксировались клетки доски, находящиеся под боем хотя бы одной из фигур — ферзя и/или ладьи.

Задания для самостоятельной работы

1. Дан двумерный массив. Найдите в каждом столбце количество и сумму элементов:
   - кратных a или b;
   - попадающих в интервал от a до b;

- являющихся простыми числами;
- положительных и лежащих выше главной диагонали.

2. Дан двумерный массив. Найдите:
   - сумму элементов в строках с k_1 по k_2;
   - номера всех максимальных элементов;
   - номера первых отрицательных элементов каждой строки (столбца);
   - номера последних отрицательных элементов каждой строки (столбца);
   - количество элементов в каждой строке, больших (меньших) среднего арифметического элементов данной строки;
   - номера первой пары неравных элементов в каждой строке.
3. Даны два квадратных массива A и B. Выведите тот из них, у которого след меньше (сумма элементов главной диагонали).
4. Дан двумерный массив. Определите:
   - есть ли в данном массиве отрицательный элемент;
   - есть ли два одинаковых элемента;
   - есть ли данное число a среди элементов массива;
   - есть ли в заштрихованной области массива $n \times n$ (рис. 1.17) элемент, равный a:

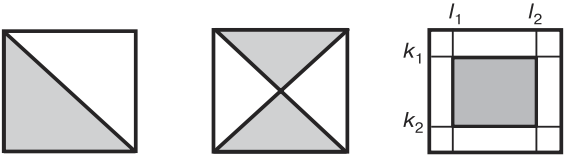


Рис. 1.17. Схематическое изображение областей двумерного массива

5. Дан двумерный массив. Определите, есть ли в данном массиве строка (столбец):
   - состоящие только из положительных элементов;
   - состоящие только из положительных или нулевых элементов;
   - состоящие только из элементов, больших числа a;
   - состоящие только из элементов, принадлежащих промежутку от a до b.

6. Дан двумерный массив. Выполните следующие его преобразования:
   - в каждой строке замените знак максимального по модулю элемента на противоположный;
   - последний отрицательный элемент каждого столбца замените нулем;
   - положительные элементы умножьте на первый элемент соответствующей строки, а отрицательные — на последний;
   - замените все элементы строки с номером k и столбца с номером l на противоположные по знаку;
   - к элементам столбца с номером l_1 прибавьте элементы столбца l_2.
7. Напишите программу, определяющую поля шахматной доски, находящиеся под боем коня при его заданном начальном положении.
8. Введите координаты ферзя и коня и определите, бьет ли одна из фигур другую.
9. Составьте программу заполнения и вывода в файл таблиц сложения и умножения цифр в заданной (произвольной) системе счисления. Для десятичной системы счисления они представлены в табл. 1.21 и табл. 1.22.

Таблица 1.21

+	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

Таблица 1.22

*	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

10. Даны два двумерных массива одинаковой размерности. Создайте третий массив той же размерности, каждый элемент которого равен:

- сумме соответствующих элементов первых двух;
- единице (True), если соответствующие элементы массивов имеют одинаковый знак и нулю (False) в противном случае.

11. Дан двумерный массив. Сформируйте одномерный массив, каждый элемент которого равен:

- произведению положительных четных элементов соответствующего столбца;
- количеству элементов соответствующей строки, больших данного числа;
- наибольшему по модулю элементу соответствующего столбца;
- количеству отрицательных элементов соответствующей строки, кратных 3 или 5;
- первому четному элементу соответствующего столбца либо нулю, если такого элемента нет.

12. Дан двумерный массив. Определите, есть ли в нем:

- строка, содержащая ровно два отрицательных элемента;
- столбец, содержащий одинаковые элементы;
- строка, содержащая два максимальных элемента всего массива;

- столбец, содержащий равное количество положительных и отрицательных элементов;
- строка, содержащая больше положительных элементов, чем отрицательных.

13. Сформируйте двумерные массивы по следующим правилам:

а) $A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$; б) $A = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$;

в) $A = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$; г) $A = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 3 & 6 & 10 & 15 & 21 \\ 1 & 4 & 10 & 20 & 35 & 56 \\ 1 & 5 & 15 & 35 & 70 & 126 \\ 1 & 6 & 21 & 56 & 126 & 252 \end{pmatrix}$;

д) $A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 3 & 4 & 5 & 6 & 1 \\ 3 & 4 & 5 & 6 & 1 & 2 \\ 4 & 5 & 6 & 1 & 2 & 3 \\ 5 & 6 & 1 & 2 & 3 & 4 \\ 6 & 1 & 2 & 3 & 4 & 5 \end{pmatrix}$; е) $A = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 20 & 21 & 22 & 23 & 24 & 7 \\ 19 & 32 & 33 & 34 & 25 & 8 \\ 18 & 31 & 36 & 35 & 26 & 9 \\ 17 & 30 & 29 & 28 & 27 & 10 \\ 16 & 15 & 14 & 13 & 12 & 11 \end{pmatrix}$.

14. Дан массив ($n \times n$), все значения элементов которого различны и лежат в интервале от 1 до ($n \cdot n$). Определите, является ли массив магическим квадратом, т. е. совпадают ли суммы его элементов по всем горизонталям, вертикалям и двум диагоналям.

Примеры магических квадратов:

$$\begin{pmatrix} 1 & 15 & 24 & 8 & 17 \\ 9 & 18 & 2 & 11 & 25 \\ 12 & 21 & 10 & 19 & 3 \\ 20 & 4 & 13 & 22 & 6 \\ 23 & 7 & 16 & 5 & 14 \end{pmatrix}, \quad \begin{pmatrix} 38 & 14 & 32 & 1 & 26 & 44 & 20 \\ 5 & 23 & 48 & 17 & 42 & 11 & 29 \\ 21 & 39 & 8 & 33 & 2 & 27 & 45 \\ 30 & 6 & 24 & 49 & 18 & 36 & 12 \\ 46 & 15 & 40 & 9 & 34 & 3 & 28 \\ 13 & 31 & 7 & 25 & 43 & 19 & 37 \\ 22 & 47 & 16 & 41 & 10 & 35 & 4 \end{pmatrix}.$$

Поскольку сумма всех элементов магического квадрата есть сумма членов арифметической прогрессии от 1 до n^2 и, следовательно, равна $n^2 \cdot (n^2+1)/2$, то суммы по строкам, столбцам и диагоналям в силу их одинаковости всегда в n раз меньше, т. е. равны $n \cdot (n^2+1)/2$. Так, при $n=5$ эти суммы равны $5 \cdot (5^2+1)/2=65$.

15. Дан двумерный массив ($n \times n$). Зеркально отобразите его элементы относительно:

- горизонтальной оси симметрии;
- вертикальной оси симметрии;
- главной диагонали;
- побочной диагонали.

Занятие № 18. Двумерные массивы. Вставка и удаление

План занятия

1. Эксперименты с программами (вставка и удаление элементов двумерного массива).
2. Выполнение самостоятельной работы.

Экспериментальный раздел занятия

1. Требуется вставить в двумерный массив строку из нулей после строки с номером t. Для решения этой задачи необходимо:

- первые t строк оставить без изменения;
- все строки после t-й сдвинуть на одну;
- элементам строки $t+1$ присвоить заданные значения.

В процедуре ввода данных кроме ввода размеров массива следует предусмотреть ввод номера строки — ReadLn(n, m, t). Ключевая процедура вставки строки имеет вид:

```
Procedure InsertStr(v,w,r:Integer;
                    Var X:TMyArray);
 Var i,j:Integer;
 Begin
   For i:=v DownTo r+1 Do
    For j:=1 To w Do X[i+1,j]:=X[i,j];
   For j:=1 To w Do X[r+1,j]:=0;
 End;
```

Измените процедуру так, чтобы новая строка из нулей вставлялась перед строкой с заданным номером.

2. Требуется вставить в двумерный массив строку из нулей после каждой строки, содержащей максимальный элемент массива. На этот раз, чтобы напомнить о стандартно используемой схеме решения задачи, приведем полный текст решения.

```
{$R+}
Program My18_1;
 Const MaxN=8; MaxM=9;
 Type TMyArray=Array[1..2*MaxN,1..MaxM]
               Of Integer;
{Резервируем 2*MaxN строк на тот случай, если
максимальный элемент массива есть в каждой
строке.}
 Var A:TMyArray;
     n,m:Integer;
 Procedure TInit(Var v,w:Integer;Var X:TMyArray);
  Var i,j:Integer;
  Begin
   Assign(Input,'Input.Txt'); Reset(Input);
   ReadLn(v,w);
   For i:=1 To v Do
    For j:=1 To w Do Read(X[i,j]);
   Close(Input);
  End;
 Procedure TPrint(v,w:Integer;X:TMyArray);
  Var i,j:Integer;
  Begin
   Assign(Output,'Output.Txt'); ReWrite(Output);
```

```
  For i:=1 To v Do Begin
   For j:=1 To w Do Write(X[i,j]:4);
   WriteLn;
  End;
  Close(Output);
 End;
Procedure InsertStr(v,w,r:Integer;
                    Var X:TMyArray);
 Var i,j:Integer;
 Begin
  For i:=v DownTo r+1 Do
   For j:=1 To w Do X[i+1,j]:=X[i,j];
  For j:=1 To w Do X[r+1,j]:=0;
 End;
Function TMax(v,w:Integer;X:TMyArray): Integer;
             {Поиск максимального элемента.}
 Var i,j,max:Integer;
 Begin
  max:=X[1,1];
  For i:=1 To v Do
   For j:=1 To w Do
    If X[i,j]>max Then max:=X[i,j];
  TMax:=max;
 End;
Procedure Solve(Var v:Integer;w:Integer;
                Var X:TMyArray);
 Var i,j,smax:Integer;
 Begin
  smax:=TMax(v,w,X); {Находим максимальный
                      элемент.}
  i:=1;
  While i<=v Do Begin
   j:=1;
   While (j<=w) And (X[i,j]<>smax) Do j:=j+1;
   If j<>w+1 Then Begin {В строке есть
                         максимальный элемент.}
     InsertStr(v,w,i,X); {Вставляем строку.}
     v:=v+1; {Увеличиваем количество строк.}
     i:=i+2; {Пропускаем вставленную строку.}
    End
   Else i:=i+1; {Обычное изменение индекса.}
```

```
  End;
 End;
Begin {В основной программе только вызовы
       процедур – крупных «кирпичиков».}
 TInit(n,m,A);
 Solve(n,m,A);
 TPrint(n,m,A);
End.
```

Измените решение для вставки нулевых строк перед строками, содержащими максимальный элемент массива.

3. Удалить строку с номером t из данного массива.

Для удаления строки с номером t необходимо сдвинуть все строки, начиная с $t+1$, на одну вверх и последнюю строку «обнулить» либо не рассматривать как строку массива.

```
Procedure DelStr(v,w,t:Integer;Var X:TMyArray);
 Var i,j:Integer;
 Begin
  For i:=t To v-1 Do
   For j:=1 To w Do X[i,j]:=X[i+1,j];
  For j:=1 To w Do X[v,j]:=0;
 End;
```

4. Удалить строки, содержащие максимальный элемент массива. Задача решается следующей процедурой:

```
Procedure Solve(Var v:Integer;w:Integer;
                Var X:TMyArray);
 Var i,j,smax:Integer;
 Begin
  smax:=TMax(v,w,X); {Находим максимальный
                      элемент.}
  i:=1;
  While i<=v Do Begin
   j:=1;
   While (j<=w) And (X[i,j]<>smax) Do j:=j+1;
   If j<>w+1 Then Begin
      DelStr(v,w,i,X);
      v:=v-1;
    End
    Else i:=i+1;
 End;
End;
```

Сравните эту процедуру с процедурой Solve, рассмотренной ранее при вставке строк, и объясните отличия.

5. Поменять местами элементы столбцов с номерами l_1 и l_2.

С процедурой Swap мы уже встречались. Напомним ее.

```
Procedure Swap(Var x, y: Integer);
 Var z: Integer;
 Begin
  z:=x;
  x:=y;
  y:=z;
 End;
```

Написание процедуры, реализующей требования примера, не вызывает затруднений.

```
Procedure Change(v,st1,st2:Integer;
                 Var X:TmyArray);
 Var i:Integer;
 Begin
  For i:=1 To v Do Swap(X[i,st1],X[i,st2]);
 End;
```

6. Удалить все строки и столбцы, содержащие максимальный элемент массива.

Вначале рассмотрим пример:

$$A=\begin{pmatrix} 2 & 7 & 9 & 3 & 4 \\ 5 & 6 & 9 & 0 & 1 \\ 2 & 1 & 9 & 5 & 8 \\ 7 & 6 & 9 & 4 & 3 \\ 5 & 6 & 9 & 2 & 1 \end{pmatrix}.$$

Максимальный элемент равен 9. Если выбрать логику просмотра по строкам, а затем по столбцам, то результат окажется верным. При просмотре же вначале по столбцам, а затем по строкам исключается только третий столбец. Значит, надо вначале запомнить номера строк и столбцов, содержащих максимальный элемент массива, и лишь затем их удалять.

Пусть в процессе просмотра начального массива мы запомнили в массивах *NumStr* и *NumStl* номера удаляемых строк и столбцов. Для рассматриваемого примера они имеют вид: (1, 2, 3, 4, 5) и (3, 3, 3, 3, 3). Очевидно, что пять раз удалять третий

столбец нет смысла, поэтому из массивов необходимо удалить повторяющиеся элементы, то есть во втором массиве оставить один элемент 3.

Однако этим не ограничиваются возникающие сложности. Первая строка удалена, требуется удалять вторую, но она уже в нашей матрице стала первой. Значит, необходимо после каждого удаления изменить номера строк (столбцов) в массиве *NumStr* — уменьшить на единицу те номера, которые больше номера удаленной строки. Ниже по тексту приведена только процедура Solve, остальные части программы совпадают с ранее рассмотренными.

```
Procedure Solve(Var v,w:Integer;Var X:TMyArray);
 Type OMyArray=Array[1..MaxN] Of Integer;
    {Определяем одномерный массив для хранения
    номеров удаляемых элементов. Константа MaxN
    берется в предположении, что количество строк
    в матрице всегда больше количества столбцов.}
 Var i,smax:Integer;
    NumStr,NumStl:OMyArray; {Номера удаляемых
                             строк и столбцов.}
    ykStr,ykStl:Integer; {Количество элементов
                  в массивах NumStr и NumStl.}
 Procedure FNum(Var q,r:Integer;
                Var Y,Z:OMyArray); {Формирование
                массивов с номерами удаляемых
                строк и столбцов.}
  Var i,j:Integer;
  Begin
   q:=0;
   r:=0;
   For i:=1 To v Do
    For j:=1 To w Do
     If X[i,j]=smax Then Begin
       q:=q+1;
       Y[q]:=i;
       r:=r+1;
       Z[r]:=j;
      End;
  End;
 Procedure Sz(Var t:Integer; Var X:OMyArray);
{Удаление из массива повторяющихся элементов.}
```

```
 Var i,j,l:Integer;
 Begin
  i:=1;
  While i<t Do Begin
   j:=i+1;
   While j<=t Do
    If X[i]=X[j] Then Begin
       For l:=j To t-1 Do X[l]:=X[l+1];
       X[t]:=0;
       t:=t+1;
      End
     Else j:=j+1;
   i:=i+1;
  End;
 End;
Procedure Change(t,a:Integer;Var X:OMyArray);
     {Уменьшение на единицу значений элементов
     массива, превышающих заданную величину.}
 Var i:Integer;
 Begin
  For i:=1 To t Do
   If X[i]>a Then X[i]:=X[i]-1;
 End;
Begin {Текст основной процедуры.}
 FillChar(NumStr,SizeOf(NumStr),0);
       {Инициализация массивов.}
 FillChar(NumStl,SizeOf(NumStl),0);
 smax:=TMax(v,w,X); {Находим максимальное
       значение – текст функции не приводится.}
 FNum(ykStr,ykStl,NumStr,NumStl); {Определяем
       номера удаляемых строк и столбцов.}
 Sz(ykStr,NumStr); {Удаляем повторяющиеся
                    элементы.}
 Sz(ykStl,NumStl);
 i:=1;
 While i<=ykStr  Do Begin
   DelStr(v,w,NumStr[i],X); {Удаляем строку.}
   Change(ykStr,NumStr[i],NumStr); {Корректируем
                     массив номеров строк.}
   v:=v-1;
   i:=i+1;
```

```
End;
i:=1;
While i<=ykStl  Do Begin
  DelStl(v,w,NumStl[i],X); {Удаляем столбец.}
  Change(ykStl,NumStl[i],NumStl); {Корректируем
                    массив номеров столбцов.}
  w:=w-1;
  i:=i+1;
 End;
End;
```

О технологии написания. Напоминаем, что процесс разработки процедуры Solve заключается в написании «тела» процедуры с заголовками составляющих частей. Программа компилируется, сохраняется и только затем дописываются составляющие части. В процедуре Solve есть два практически совпадающих фрагмента (циклы по *i*). Исключите эти повторения.

Задания для самостоятельной работы

1. Вставьте в двумерный массив:

- первую строку после строки, содержащей первый встреченный минимальный элемент;
- второй столбец после первого встреченного столбца, в котором все элементы положительные;
- нулевую строку и нулевой столбец перед строкой и столбцом, в которых находится первый минимальный элемент;
- последнюю строку после каждой строки, содержащей заданное число *a*;
- второй столбец перед каждым столбцом, не содержащим отрицательных элементов;
- первую строку перед каждой строкой, содержащей 0, и первый столбец после каждого столбца, содержащего отрицательные элементы;
- столбец из нулей после каждого столбца с минимальными элементами;
- первую строку между средними строками (количество строк четное);
- строку из нулей перед каждой строкой, первый элемент которой делится на три.

2. Удалите:
- столбцы, в которых есть минимальный элемент;
- строку с номером k и столбец с номером l;
- все столбцы, в которых нет нулевого элемента;
- все строки и столбцы, на пересечении которых стоят отрицательные элементы;
- среднюю строку (строки);
- все столбцы, в которых первый элемент больше последнего;
- столбец, содержащий первый четный отрицательный элемент;
- все столбцы, в которых первый элемент больше заданного числа a.

3. Замените:
- минимальный элемент в каждой строке на противоположный по знаку;
- все элементы первых трех столбцов на их квадраты.

4. Поменяйте местами:
- средние столбцы (количество столбцов четное);
- средние строки с первой и последней (количество строк четное);
- средние столбцы со вторым и предпоследним (количество столбцов четное);
- средние строки (количество строк четное);
- первый максимальный и последний минимальный элементы;
- в каждой строке первый элемент и максимальный по модулю;
- в каждой строке первый отрицательный элемент и последний положительный;
- первую строку и строку, содержащую первый нулевой элемент;
- вторую и предпоследнюю строки;
- первую строку с последней строкой, вторую — с предпоследней и так далее;
- столбцы в следующем порядке:
  а) последний, предпоследний, …, второй, первый;
  б) первый, последний, второй, предпоследний, третий, …

5. Найдите максимальный элемент массива, встречающийся более одного раза.

6. Для целочисленного массива ($n \times n$) найдите максимум среди сумм элементов диагоналей, параллельных главной диагонали.
7. Для целочисленного массива ($n \times n$) найдите минимум среди сумм элементов диагоналей, параллельных побочной диагонали.
8. Значения элементов массива находятся в интервале от a до b. Две строки матрицы назовем похожими, если они отличаются только порядком элементов. Найдите пары похожих строк.
9. Элемент массива $A[i,j]$ называют седловой точкой, если он является минимальным в строке с номером i и максимальным в столбце с номером j. Найдите седловые точки.
10. Для данного массива ($n \times n$) найдите такие значения k, что k-я строка совпадает с k-м столбцом.
11. В целочисленном массиве найдите столбцы, состоящие только из нечетных элементов.
12. Элемент $A[i,j]$ назовем локальным минимумом, если он строго меньше всех своих соседей. Соседями являются элементы $A[k,l]$ с $(i-1 \le k \le i+1)$, $(j-1 \le l \le j+1)$, $(k,l) \ne (i,j)$. Найдите количество локальных минимумов для заданного массива. Найдите максимум среди локальных минимумов.
13. Для двумерного массива $A(n \times m)$ (элементы в строках и столбцах которого упорядочены по возрастанию или по убыванию) решите задачу поиска числа x за время, пропорциональное $O(n+m)$.
14. В квадратной матрице A порядка n переставьте строки так, чтобы элементы в первом столбце были упорядочены по неубыванию.
15. В матрице A порядка $n \times m$ замените нулями элементы, стоящие в строках и столбцах, где имеются нули.
16. Дана квадратная матрица A порядка n, каждый элемент которой равен 0, 1, 5 или 11. Подсчитайте в ней количество четверок элементов $A[i,j]$, $A[i+1,j]$, $A[i,j+1]$, $A[i+1,j+1]$, в каждой из которых все элементы различны.

    Примечание

    Если все числа различны, то их сумма равна 17. Верно и обратное утверждение.
17. Дана квадратная матрица A порядка n, состоящая только из 0 и 1. Прямоугольником назовем прямоугольную часть

матрицы, заполненную единицами. Известно, что прямоугольники не соприкасаются друг с другом. Найдите количество прямоугольников. Так, например, в матрице

$$A=\begin{pmatrix}0&1&0&0&1&1\\0&0&0&0&1&1\\1&1&0&0&0&0\\1&1&0&1&1&0\\1&1&0&1&1&0\end{pmatrix}$$

четыре прямоугольника.

18. Дана квадратная матрица A порядка n натуральных чисел. Находится минимальный элемент A и вычитается из всех элементов матрицы. Затем строки и столбцы, где после вычитания появились нули, «вычеркиваются» из матрицы. Процесс продолжается до тех пор, пока не будет получено одно число. Найдите сумму минимальных элементов матрицы.

19. Дана квадратная матрица A порядка n из 0 и 1. Считаем, что столбец «покрывается» другим столбцом, если множество строк, представленных единицами в этом столбце, содержится в множестве строк, представленных единицами в другом столбце. Исключите такие столбцы из матрицы.

20. Дана квадратная матрица A порядка n из 0 и 1. Исключите из матрицы столбцы и строки, содержащие одну-единственную единицу. Если после исключения появились новые строки и столбцы, удовлетворяющие этому условию, то продолжите процесс исключения.

21. Квадратную матрицу A порядка n из 0 и 1 назовем правильной, если в ней нет квадратов 2×2 и более, составленных только из 0 или 1. Проверьте, является ли A правильной. Так, например,

$$A=\begin{pmatrix}1&0&1&0&1\\0&1&0&1&0\\1&0&1&0&1\\0&1&1&0&1\\1&0&0&0&1\end{pmatrix}$$

является правильной.

1.4. Дополнительные занятия

Занятие № 19. Вещественный тип данных

План занятия

1. Вещественный тип данных.
2. Короткие программы иллюстрации работы с вещественным типом данных.
3. Эксперименты с программами (решение уравнения $f(x)=0$; вычисление площади фигуры; вычисление элементарных функций с использованием представления этих функций в виде некоторых бесконечных сумм).
4. Выполнение самостоятельной работы.

Вещественный тип данных

Вещественные числа записываются

- путем отделения дробной части от целой с помощью точки, например 127.3, 25.0, –16.003, 200.59, 0.54;
- с использованием символа E, например:

0,000009	9E–6
$0{,}62 \cdot 10^4$	0.62E+4
$-10{,}8 \cdot 10^{12}$	–10.8E12
$20 \cdot 10^{-3}$	20E–3

Вначале записывается мантисса (дробная часть числа), затем порядок (степень десяти, обозначается буквой E). Существуют следующие типы вещественных чисел (наиболее используемый — Real).

Таблица 1.23

Тип	Диапазон возможных значений	Точность	Формат
Real	2.9E–39...1.7E38	11–12 знаков	6 байт
Single	1.5E–45...3.4E38	7–8 знаков	4 байта
Double	5.0E–324...1.7E308	15–16 знаков	8 байт
Extended	3.4E–4932...1.1E4932	19–20 знаков	10 байт
Comp	–9.2E18...9.2E18	19–20 знаков	8 байт

Примечание

Величины типа данных Comp принимают целочисленные значения, в известной степени этот тип является расширением типа LongInt до 19 разрядов.

Допустимые арифметические операции: «+» сложение, «–» вычитание, «*» умножение, «/» деление и операции сравнения. На вещественном типе данных нет отношения порядка, поэтому операцией сравнения «=» следует пользоваться с большой осторожностью.

Перечислим стандартные функции работы с вещественными величинами (табл. 1.24).

Таблица 1.24

Sqr(X)	Квадрат *X*
Sqrt(*X*)	Квадратный корень из *X*
Sin(*X*)	Синус *X* (аргумент задается в радианах)
Cos(*X*)	Косинус *X* (аргумент задается в радианах)
Arctan(*X*)	Арктангенс *X* (результат в радианах)
Exp(*X*)	Число *e* в степени *X*
Ln(*X*)	Натуральный логарифм *X*
Pi	Значение числа π
Int(*X*)	Целая часть *X* (результат представлен в формате Integer)
Frac(*X*)	Дробная часть *X*
Tranc(*X*)	От аргумента «отбрасывается» дробная часть: Tranc(6.8)=6, Tranc(–7.9)=7 (результат представлен в формате LongInt)
Round(*X*)	Ближайшее к *X* целое число: Round(5.8)=6, Round(5.5)=6

Короткие программы

Первый пример связан со сравнением двух вещественных чисел. Запустите эту простую программу.

```
Program My19_1;
 Var x:Real;
     Y:Real;
 Begin
  x:=0.3;
  y:=0.3;
  WriteLn(x=y);
  ReadLn;
 End.
```

Вы получите ожидаемый результат — значение True. Измените описание переменной *y* на `y:Single`. При запуске программы возникает ошибка: `Error 116: Must be in 8087`

mode to compile this[11]. Суть в том, что выполнение арифметических операций с вещественными числами осуществляется иначе, чем с целыми числами. Их реализация требует больших затрат процессорного времени. С этой целью созданы специальные быстродействующие математические сопроцессоры.

Директива {$N+}.
Данная директива компилятора осуществляет переключение между двумя различными режимами генерации кода для выполнения операций с вещественными числами. В режиме {$N–}, а по умолчанию включен он, компилятор использует библиотеку подпрограмм для работы с 6-байтовыми вещественными числами, то есть типом Real. В режиме {$N+} компилятор генерирует программный код, обеспечивающий повышенную точность вычислений и доступ к четырем дополнительным вещественным типам данных.

Итак, при использовании дополнительных вещественных типов данных требуется записывать директиву {$N+} в начало программы. Вставим ее и добавим еще два оператора.

```
{$N+}
Program My19_1m;
 Var x:Real;
     y:Single;
 Begin
  x:=0.3;
  y:=0.3;
  WriteLn(x=y);
  WriteLn(x);
  WriteLn(y);
  ReadLn;
 End.
```

Результат сравнения есть False. Это происходит потому, что для хранения значений *x* и *y* выделяется разное количество разрядов.

---

[11] Только в случае, если на компьтере не установлен математический сопроцессор и не включен эмулятор сопроцессора.

Отметим еще один момент. При работе в режиме {$N+} стандартная процедура Write осуществляет вывод в формате Extended, так же как и значения, возвращаемые стандартными функциями, приведенными в табл. 1.24.

Известно, что $(x/a) \cdot a = x$. Проверим это утверждение с помощью следующей простой программы.

```
{$N+}
Program My19_2;
 Var x:Extended;
     i,cnt:Integer;
 Begin
  cnt:=0;
  For i:=20000 To 30000 Do Begin
   x:=i/10000;
   x:=x*10000;
   If x=i Then cnt:=cnt+1;
  End;
  WriteLn('Число совпадений ',cnt);
  WriteLn('Число несовпадений ',10001-cnt);
  ReadLn;
 End.
```

Результат. Число совпадений — 8209, число несовпадений — 1792. Попытка вывести значения x и i при несовпадениях ничего не дает. При выводе числа кажутся равными, то есть они различаются в разрядах, которые не используются при выводе значений переменных на экран. Мы постараемся не использовать непосредственного сравнения величин вещественного типа, а будем применять для этих целей короткую функцию. Она приведена в следующей модификации программы.

```
{$N+}
Program My19_3;
 Const eps=1.0E-10;
Var x:Extended;
    i,cnt:Integer;
 Function Eq(x,y:Real):Boolean;
Begin
   Eq:=Abs(x-y)<eps;
  End;
```

```
Begin
 cnt:=0;
 For i:=20000 To 30000 Do Begin
  x:=i/10000;
  x:=x*10000;
  If Eq(x,i) Then cnt:=cnt+1;
 End;
 WriteLn('Число совпадений ',cnt);
 WriteLn('Число несовпадений ',10001-cnt);
 ReadLn;
End.
```

Результат работы программы — 1001 совпадение.

Экспериментальный раздел занятия

1. Рассмотрим часто встречающуюся задачу приближенного решения уравнения $f(x)=0$, где $f(x)$ — заданная функция. Решить уравнение — значит, найти такое значение $x=c$, при котором $f(c)=0$. Поиск решения ведется на интервале $[a, b]$, причем $f(a)<0$, а $f(b)>0$. Поиск такого интервала — отдельный вопрос, мы его не рассматриваем. Для решения используем одну из фундаментальных идей информатики — «разделяй и властвуй», в данном случае метод деления пополам. Находим точку c — половину отрезка $[a, b]$. Если $f(c)>0$, то границу b изменяем на значение c, а если $f(c)<0$, то изменяем a. Процесс продолжаем, пока длина интервала не окажется меньше заданной точности.

Пусть $f(x)=x^2-2$, то есть мы попытаемся найти значение квадратного корня из двух.

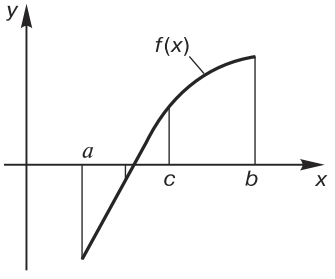


Рис. 1.18. Иллюстрация метода половинного деления

Программа будет выглядеть так:

```
{$N+}
Program My19_4;
 Const eps=1.0E-3;
 Var a,b,c:Real;
 Function Eq(x,y:Real):Boolean; {Функция
                                  рассмотрена выше.}
 Function F(x:Real):Real;
  Begin
   F:=Sqr(x)-2;
  End;
 Begin
  WriteLn('Введите интервал.');
  ReadLn(a,b);
  If F(a)*F(b)>0 Then WriteLn('На этом интервале
            мы не можем найти решение уравнения.')
   Else Begin
    While Not Eq(a,b) Do Begin
     c:=(a+b)/2;
     If F(c)>0 Then b:=c Else a:=c;
    End;
    WriteLn('Результат ',a);
   End;
  ReadLn;
 End.
```

Обратите внимание на то, что вычисление $f(x)$ оформлено как отдельная функция. Таким образом, чтобы использовать программу для решения другого уравнения, требуется изменить лишь одну функцию.

Запуск программы при различных значениях a и b вас не очень обрадует. Получаются значения 1.4140625 или 1.41357421875 (для различных интервалов a и b), а фактическое значение равно 1.414213...

Измените значение *eps* на 1.0E–6. Точность вычислений имеет большое значение при работе с данными вещественного типа.

Решите с помощью приведенной программы еще ряд уравнений:

- $x^2-6x+5=0$;
- $x-\cos(x)=0$;

- $x-\ln(x)-2=0$;
- $2x^3-9x^2-60x+1=0$.

Возникает вопрос: как определять интервал $[a, b]$, удовлетворяющий условиям задачи? Начнем с составления таблицы. Вначале составим ее вручную, например, для первой функции $x^2-6x+5=0$ — табл. 1.25.

Таблица 1.25

X	0	2	3	6
Знак $f(x)$	+	–	–	+

После нескольких попыток можно понять алгоритм, который лежит в основе нахождения интервала $[a, b]$. Добавьте к программе еще одну процедуру, позволяющую решить эту задачу.

2. Пусть $f(x)$ — непрерывная положительная функция на отрезке $[a, b]$ $(a<b)$. Вычислим площадь фигуры, ограниченную графиком функции $f(x)$, осью x и прямыми $x=a$ и $x=b$. Для этого разобъем отрезок $[a, b]$ на n равных отрезков одинаковой длины $\Delta x = \dfrac{b-a}{n}$: $a=x(0)<x(1)<...<x(i)<x(i+1)<...< x(n)=b$.

На каждом из отрезков $[x(i), x(i+1)]$ как на основании построим прямоугольник с высотой $f(x(i))$. Площадь прямоугольников мы умеем находить. Сумма площадей всех прямоугольников даст приближенное значение площади фигуры (рис. 1.19):

$$S_{прибл.} = \frac{b-a}{n}\cdot(f(x(0)) + f(x(1))+...+f(x(n-1))).$$

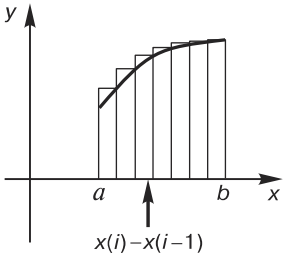


Рис. 1.19. Схема вычисления площади фигуры

Естественно, что чем «мельче» будет разбиение, тем точнее мы подсчитаем площадь фигуры.

Примечание

Найдите неточность в объяснении логики вычисления площади.

Для отладки программного решения возьмем простую функцию $f(x)=x^2$ на интервале от 1 до 2. Площадь фигуры равна 2.333.

```
{$N+;}
Program My19_5;
 Const eps=1.0E-3;
 Var a,b,s,dx:Real;
     i,n:Integer;
 Function Eq(x,y:Real):Boolean; {Функция
                                 рассмотрена выше.}
 Function F(x:Real):Real;
  Begin
   F:=Sqr(x);
  End;
 Begin
  WriteLn('Введите интервал и число частей,
          на которые он разбивается');
  ReadLn(a,b,n);
  x:=a;
  dx:=(b-a)/n;
  s:=0;
  For i:=1 To n-1 Do Begin
   s:=s+F(x);
   x:=x+dx;
  End;
  WriteLn('Результат ',(b-a)/n*s);
  ReadLn;
 End.
```

Запускаем программу при фиксированных границах интервала и различных значениях n. Результаты удручают. При n=6 он равен 1.5277..., при n=50 — 2.224992..., при n=100 — 2.78749..., при n=300 — 2.315046... Давайте изменим процедуру решения. Будем сравнивать два соседних приближенных значения площади для различных n. Если их разность больше

заданной точности вычисления, то увеличим значение n и снова подсчитаем площадь. Начальное значение n в программе взято 10, шаг изменения 100. Естественно, эти параметры программы вынесены в раздел констант, так же как и точность вычисления.

```
{$N+;}
Program My19_5m;
 Const eps=1.0E-3;
       nb=10;ns=100;
 Var a,b,wn,ws:Real;
     i,n:Integer;
 Function Eq(x,y:Real):Boolean; {Функция
                          рассмотрена выше.}
 Function F(x:Real):Real;
 Function Sq(n:Integer):Real;
  Var x,dx,s:Real;
  Begin
   x:=a;
   dx:=(b-a)/n;
   s:=0;
   For i:=1 To n-1 Do Begin
    s:=s+F(x);
    x:=x+dx;
   End;
   Sq:=(b-a)/n*s;
  End;
 Begin
  WriteLn('Введите границы интервала');
  ReadLn(a,b);
  n:=nb;
  wn:=Sq(n);
  Repeat
   ws:=wn;
   n:=n+ns;
   wn:=Sq(n);
  Until Eq(ws,wn);
  WriteLn('Результат: количество частей
          и значение площади',n,wn);
  ReadLn;
 End.
```

После запуска увидите, что при n=810 достигается требуемая точность вычислений: результат — 2.32654955…. Однако в уме мы подсчитали точнее. Измените точность на 1.0E–6. Компьютер задумается на некоторое время, а результат окажется таким: n=23310 и площадь — 2.33309739….

Вычислите площадь криволинейных трапеций для следующих функций:

$f(x)=\frac{1}{1+x}$, интервал [0,1];

$f(x)=\frac{1}{x}$, интервал [1,3];

$f(x)=\sin(x)$, интервал [0, $\pi/2$].

3. Для вычисления элементарных функций в математике широко распространено представление этих функций в виде некоторых бесконечных сумм. Не вдаваясь в обоснование таких представлений, приведем некоторые из них:

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + \ldots;$$

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

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

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots + (-1)^{n+1} \frac{x^n}{n} + \ldots, (-1 < x \le 1).$$

В каждом из разложений точность представления функции будет, вообще говоря, тем выше, чем больше взято слагаемых в сумме. Причем значения самих слагаемых с ростом n стремятся к нулю. Для вычисления значений функции с некоторой заданной точностью *eps* поступают следующим образом. Вычисляют и суммируют слагаемые до тех пор, пока очередное слагаемое не станет по абсолютной величине меньше *eps* или абсолютное значение разности между соседними слагаемыми не станет меньше *eps* (что лучше?). Полученную сумму и принимают за приближенное значение функции. Продемонстрируем этот метод на примере вычисления функции $\sin(x)$.

```
{$N+}
Program My19_6;
```

```
Const eps=1.0E-3;
Var x,sn,ss:Real;
    p,n:Integer;
Function Eq(x,y:Real):Boolean; {Функция
                  рассмотрена выше.}
Function F(n:Integer;Var x:Real):Real;
 Var i:Integer;
     s:LongInt;
 Begin
  s:=1;
  For i:=2 To n Do s:=s*i;
  x:=x*Sqr(x);
  F:=x/s;
 End;
Begin
 WriteLn('Введите значение x ');
 ReadLn(x);
 ss:=0;
 sn:=x;
 n:=1;
 p:=1; {p - для реализации чередования знака
       слагаемого.}
 Repeat
  ss:=sn; {Предыдущее значение слагаемого.}
  n:=n+2;
  p:=-1*p;
  sn:=ss+p*F(n,x); {Новое значение слагаемого.}
 Until Eq(ss,sn) Or (n>=12); {Второе условие
   требуется по простой причине - факториал числа
   мы умеем вычислять только при этих
    значениях n.}
 WriteLn('Результат ',sn);
 ReadLn;
End.
```

Известно, что $\sin(30°)=\frac{1}{2}$, $\sin(45°)=\frac{\sqrt{2}}{2}$ и что 30°=0,5236... радиан, 45°=0,7854... радиан. Проверьте работоспособность программы.

Измените программу для вычисления остальных приведенных выше функций. Исследуйте решения.

Задания для самостоятельной работы

1. Квадратное уравнение $a \cdot x^2 + b \cdot x + c = 0$ задается своими коэффициентами. Его корни находятся по формуле: $x_{1,2} = \dfrac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$. Напишите программу нахождения корней квадратного уравнения.
2. Напишите программы вычисления следующих выражений:
   - $y = \left(1 - \dfrac{1}{2}\right) \cdot \left(1 - \dfrac{1}{3}\right) \cdot \ldots \cdot \left(1 - \dfrac{1}{n}\right)$, $n > 2$;
   - $y = \cos(1 + \cos(2 + \ldots + \cos(39 + \cos 40)\ldots))$;
   - $y = \sin x + \sin\sin x + \ldots + \sin\sin\ldots\sin x$ (n раз);
   - $y = \sin x + \sin x^2 + \ldots \sin x^n$;
   - $y = \sin x + \sin^2 x + \ldots + \sin^n x$;
   - $y = \dfrac{\cos 1}{\sin 1} \cdot \dfrac{\cos 1 + \cos 2}{\sin 1 + \sin 2} \cdot \ldots \cdot \dfrac{\cos 1 + \ldots + \cos n}{\sin 1 + \ldots + \sin n}$;
   - $y = |x| + x^4$;
   - $y = |x| + 4 \cdot x^3 - 7 \cdot x^2$;
   - $y = |x - 2| + 3 \cdot x^8$;
   - $y = (3 \cdot x^3 + 18 \cdot x^2) \cdot x + 12 \cdot x^2 - 5$;
   - $y = |x + 4| - |x^2 - 3 \cdot x + 6|$.
3. Вычислите приближенные значения бесконечных сумм:
   - $\dfrac{1}{1 \cdot 2} + \dfrac{1}{2 \cdot 3} + \dfrac{1}{3 \cdot 4} + \ldots$;
   - $\dfrac{1}{1 \cdot 3} + \dfrac{1}{2 \cdot 4} + \dfrac{1}{3 \cdot 5} + \ldots$;
   - $\dfrac{1}{1 \cdot 2 \cdot 3} + \dfrac{1}{2 \cdot 3 \cdot 4} + \dfrac{1}{3 \cdot 4 \cdot 5} + \ldots$.
4. Напишите программу вычисления

$$1 - \frac{1}{2} + \frac{1}{3} - \ldots + \frac{1}{9999} - \frac{1}{10000}$$

следующими четырьмя способами:

- последовательно слева направо;
- последовательно слева направо вычисляются $1+\frac{1}{3}+\ldots+\frac{1}{9999}$ и $\frac{1}{2}+\frac{1}{4}+\ldots+\frac{1}{10000}$, затем второе значение вычитается из первого;
- последовательно справа налево;
- последовательно справа налево вычисляются суммы (как во втором случае), а затем выполняется вычитание.

Сравните результаты и найдите их объяснение.

5. Рассмотрим бесконечную последовательность чисел y_1, y_2, y_3, ..., образованную по следующему закону:

$$y_1 = \frac{x+m-1}{2},$$

$$y_i = \frac{1}{m}\left((m-1)\cdot y_{i-1} + \frac{x}{y_{i-1}^{m-1}}\right),\ i = 2, 3, \ldots,$$ где x — данное действительное число, m — натуральное число. Эта последовательность позволяет получить сколь угодно точные приближения числа $\sqrt[m]{x}$ ($x \geq 0$). Составьте программу для вычисления значения $\sqrt[m]{x}$ с заданной точностью *eps*.

6. Дано n вещественных чисел. Напишите программы:

- вычисления разности между максимальным и минимальным элементами;
- определения количества элементов, которые больше обоих своих «соседей», то есть предыдущего и последующего чисел;
- определения минимального значения разности двух произвольных элементов из n.

Занятие № 20. Множественный тип данных

План занятия

1. Множественный тип данных.

2. Операции над множествами.

3. Эксперименты с программами (вывод цифр, не входящих в десятичную запись числа; поиск простых чисел с помощью «решета Эратосфена»; решение ребусов).

4. Выполнение самостоятельной работы.

Множественный тип данных

Множество в языке Паскаль — это ограниченный упорядоченный набор различных элементов одного (базового) типа. *Базовый тип* — это совокупность значений, из которых могут быть образованы множества. Мощность образуемых множеств (количество различных элементов) не превышает 256. Значение переменной множественного типа может содержать любое количество различных элементов базового типа. Иными словами, возможными значениями переменных множественного типа являются все подмножества значений базового типа. Вещественный тип данных не используется в качестве базового (нет отношения порядка, нельзя ввести ограничения на тип).

Описание множественного типа:

```
Type <имя типа> = Set Of <тип элементов>;
Var <имя переменной множественного типа> :  <имя типа>;
```

или

```
Var <имя переменной множественного типа> :  Set Of <тип элементов>;
```

Пример использования множественного типа:

```
Type Mn_Char= Set Of Char;
Var mn1 : Set Of Char;
    mn2: Mn_Char;
    mn3: Set Of 'A'..'Z';
    s1: Set Of Byte;
    s2: Set Of 1000..1200;
```

Значениями переменных $mn1$ и $mn2$ являются множества, составленные из различных символов, $mn3$ — множества из больших латинских букв; $s1$ — множества из целых чисел от 0 до 255 (тип Byte содержит целые числа от 0 до 255), $s2$ — множества из целых чисел от 1000 до 1200.

В программе элементы множества задаются в квадратных скобках, через запятую. Если элементы идут подряд друг за другом, то используется обозначение интервала (две точки).

Пример:

```
Type  Digit=Set Of 1..5;
Var  s : Digit;
```

Количество различных значений переменной s=32. Перечислим их:

- [ ] — пустое множество;
- [1], [2], [3], [4], [5] — одноэлементные множества;
- [1,2], [1,3], ..., [2,4], [4,5] — двухэлементные множества;
- [1,2,3], [1,2,4], ..., [3,4,5] — трехэлементные множества;
- [1,2,3,4], [1,2,3,5], [1,2,4,5], [1,3,4,5], [2,3,4,5] — четырехэлементные множества;
- [1,2,3,4,5] — множество из всех элементов базового типа.

Операции над множествами

Объединением двух данных множеств называется множество элементов, принадлежащих обоим множествам либо одному из них. Знак операции — «+».

Примеры:

- ['A', 'F']+['B', 'D']=['A', 'F', 'B', 'D'];
- [1..3,5,7,11]+[3..8,10,12,15..20]=[1..8,10..12,15..20];
- S_1:=[1..5,9]; S_2:=[3..7,12]; S:=S_1+S_2=[1..7,9,12];
- A_1:=['a'..'z']; A_1:=A_1+['A']; результат — A_1=['A', 'a'..'z'].

Пересечением двух данных множеств называется множество элементов, принадлежащих одновременно и первому, и второму множествам, т. е. только общие элементы. Знак операции — «*».

Примеры:

- ['A', 'F'] * ['B', 'D']=[ ], т. к. общих элементов нет;
- [1..3,5,7,11] * [3..8,10,12,15..20]=[3,5,7];
- S_1:=[1..5,9]; S_2:=[3..7,12]; S:=S_1 * S_2=[3..5].

Вычитанием двух множеств называется множество, состоящее из тех элементов первого множества, которые не являются элементами второго множества. Знак операции — «–».

Примеры:

- ['A', 'F']–['B', 'D']=['A', 'F'] — общих элементов нет;
- [1..3,5,7,11]–[3..8,10,12,15..20]=[1..2,11];
- S_1:=[1..5,9]; S_2:=[3..7,12]; S:=S_1–S_2=[1..2,9];
- A_1:=['A'..'Z']; A_1:=A_1–['A']=['B'..'Z'].

Операция определения принадлежности элемента множеству. Эта логическая операция обозначается служебным словом In. Результат операции — значение True, если есть принадлежность к множеству, и False — в противном случае.

Примеры:

- 5 In [3..7] дает значение True, так как 5 принадлежит множеству [3..7];
- 'a' In ['A'..'Z'] дает значение False, буквы 'a' нет среди прописных латинских букв.

Операцию проверки принадлежности удобно использовать для исключения более сложных проверок. Например, оператор вида:

```
If (ch='a') Or (ch='b') Or (ch='x') Or (ch='y')
Then s;
```

переписывается в более компактной и наглядной форме:

```
If ch In ['a', 'b', 'x', 'y'] Then s;
```

Сравнение множеств. Для сравнения множеств используются обычные операции:

= — равенство (совпадение) двух множеств;

<> — неравенство двух множеств;

<=, < — проверка на вхождение первого множества во второе множество;

>=, > — проверка на вхождение второго множества в первое множество.

Экспериментальный раздел занятия

1. Дано натуральное число n. Составить программу вывода цифр, не входящих в десятичную запись числа n (в порядке возрастания).

```
Program My20_1;
 Type Mn=Set Of 0..9;
 Var s:Mn;
     n:LongInt;
     i:Integer;
 Begin
  WriteLn('Введите число ');
  ReadLn(n);
  s:=[0..9];
  While n<>0 Do Begin
   s:=s-[n Mod 10]; {Исключаем цифру.}
   n:=n Div 10;
  End;
```

```
 For i:=0 To 9 Do
  If i In s Then Write(i:2);
 WriteLn;
 ReadLn;
End.
```

Измените программу так, чтобы находились общие цифры в записи m чисел.

2. «Решето Эратосфена». Найти простые числа в интервале от 2 до n.

Напомним, что простым числом называется число, не имеющее других делителей, кроме единицы и самого себя. Решение без использования множественного типа данных приведено ниже.

```
Program My20_2;
 Const n=1000; {Интервал чисел, в котором
                находятся простые числа.}
 Var i:Integer;
 Function Simple(m:Integer):Boolean;
  Var i:Integer;
  Begin
   i:=2;
   While (i<= m Div 2) And (m Mod i<>0) Do i:=i+1;
   Simple:=(i>m Div 2);
  End;
 Begin
  For i:=2 To n Do
   If Simple(i) Then Write(i,' ');
  ReadLn;
 End.
```

Измените программу так, чтобы находилась первая тысяча простых чисел.

Откажемся от этого простого решения. Применим идею «решета Эратосфена», т. к. наша цель — изучение множественного типа данных. Суть метода — считаем все числа интервала простыми, а затем «вычеркиваем» те, которые не удовлетворяют требованию простоты. Как осуществляется вычеркивание? Находим очередное невычеркнутое число, оно простое, и удаляем все числа, кратные ему. После такого «просеивания» в исходном множестве останутся только простые числа.

```
Program My20_2m;
 Const n=255;
 Type Mn=Set Of 0..n;
 Var Sim:Mn;
     i,j:Integer;
 Begin
  Sim:=[2..n];
  j:=2;
  While j<=n Div 2 Do Begin
   If j In Sim Then Begin {Поиск очередного
                           простого числа.}
    i:=j+j;
    While i<=n Do Begin {Вычеркивание.}
      Sim:=Sim-[i];
      i:=i+j;
     End;
   End;
   j:=j+1;
 End;
 For i:=2 To n Do
  If i In Sim Then Write(i:4); {Вывод оставшихся
        после вычеркивания чисел, они простые.}
 ReadLn;
End.
```

Поиск простых чисел из интервала, большего, чем 0..255, «упирается» в ограничение множественного типа данных — не более 256 значений базового типа. Уйдем от этого ограничения путем ввода массива, элементами которого являются множества. Но прежде, чем рассмотрим решение, небольшой фрагмент:

```
{$R+}
Program My20_2mm;
 Var Mn: Set Of 1..255;
     a:Word;
 Begin
  Mn:=[1..255];
  a:=258;
  If a In Mn Then WriteLn('Yes')
   Else WriteLn('No');
  ReadLn;
 End.
```

После запуска программы видим знакомую ошибку: `Error 202: Range check error`. Значение переменной *a* выходит за допустимый диапазон значений. Учтем этот факт при написании очередной версии программы.

```
{$R+}
Program My20_2mmm;
 Uses Crt;
 Const m=255;n=1000;
 Type Mn=Set Of 1..m;
      OMyArray=Array[0.. (n Div m)] Of Mn;
 Var Sim:Mn;
     A:OMyArray;
     i,j,k:Integer;
 Begin
  ClrScr;
  k:=(n Div m);
  For i:=0 To k Do A[i]:=[1..m];
  j:=2;
  While j<=n Div 2 Do Begin
   If (j Mod m) In A[j Div m] Then Begin
    i:=j+j;
    While i<=n Do Begin
      A[i Div m]:=A[i Div m]-[i Mod m];
      i:=i+j;
     End;
   End;
   j:=j+1;
  End;
  For i:=2 To n Do
   If (i Mod m) In A[i Div m] Then Write(i,' ');
  ReadLn;
 End.
```

Можно ли с помощью рассмотренного метода найти все простые числа, меньшие 1 000 000 000 ?

3. Решение ребусов. В данном случае речь идет о ребусах, в которых одинаковым буквам должны соответствовать одинаковые цифры. С использованием множественного типа данных программный код решения получается более компактным. Подсчитаем количество решений ребуса МУХА+МУХА=СЛОН.

```
Program My20_3;
 Type Mn=Set Of 0..9;
 Var i,j,cnt:Integer;
     Sm,Se:Mn;
 Procedure Change(t:Integer; Var S:Mn); {Из цифр
                      числа формируем множество.}
  Begin
   S:=[];
   While t<>0 Do Begin
    S:=S+[t Mod 10];
    t:=t Div 10;
   End;
  End;
 Function Qw(S:Mn):Integer; {Подсчитываем
                  количество элементов в множестве.}
  Var i,cnt:Integer;
  Begin
   cnt:=0;
   For i:=0 To 9 Do
    If i In S Then cnt:=cnt+1;
   Qw:=cnt;
  End;
Begin
 cnt:=0; {Счетчик числа решений.}
 For i:=1000 To 4999 Do Begin {Результат -
        четырехзначное число, поэтому слагаемое
        не превышает 4999.}
  Change(i,Sm);
  If Qw(Sm)=4 Then Begin {Если все цифры числа
         различны, то выполняем дальнейшие
         вычисления.}
   j:=2*i;
   Change(j,Se);
   If (Sm*Se=[]) And (Qw(Se)=4) Then cnt:=cnt+1;
     {Числа состоят из различных цифр, и все цифры
      результата различны.}
  End;
 End;
 WriteLn(cnt);
End.
```

Задания для самостоятельной работы

1. Дан текст. Найдите множества, элементами которых являются встречающиеся в тексте:
   - цифры от 0 до 9 и знаки арифметических операций;
   - буквы от A до F и от X до Z;
   - знаки препинания и буквы от E до N.
2. Выведите в алфавитном порядке элементы множества, составленного из произвольных букв A ... Z.
3. Дан текст. Найдите множество строчных латинских букв, входящих в него. Подсчитайте количество знаков препинания и цифр в тексте.
4. Дан текст. Выведите в алфавитном порядке все буквы текста, входящие в него:
   - не менее двух раз;
   - не более двух раз;
   - более двух раз;
   - по одному разу.
5. Дан текст. Подсчитайте количество гласных и согласных букв.
6. Напишите программы решения ребусов:
   ЛОБ+ТРИ=САМ;
   ИСК+ИСК=КСИ;
   ТОЧКА+КРУГ=КОНУС;
   VOLVO+FIAT=MOTOR;
   AB+BC+CA=ABC.

Занятие № 21. Комбинированный тип данных (записи)

План занятия

1. Комбинированный тип данных (Record).
2. Оператор выбора (Case).
3. Эксперименты с программами (работа с датами и простейшими геометрическими объектами).
4. Выполнение самостоятельной работы.

Комбинированный тип данных (Record)

При работе с массивами основное ограничение заключается в том, что каждый элемент должен иметь один и тот же тип.

Однако при решении многих задач возникает необходимость хранить и обрабатывать совокупности данных различного типа как единое целое.

Запись — это составной тип данных, содержащий набор объектов разных типов. Составляющие запись объекты называются ее полями. В записи каждое поле имеет свое собственное имя. Чтобы описать запись, необходимо указать ее имя, имена объектов, составляющих запись, и их типы. Общий вид записи таков:

```
Type  <имя записи> = Record
                  <поле 1>:<тип 1>;
                  <поле 2>:<тип 2>;
                  ...
                  <поле n>:<тип n>
                 End;
```

Пример.

Опишем дату. Она состоит из года, месяца и дня. Определив их, мы определяем дату как запись, состоящую из соответствующих полей.

```
Type year=1583..3000;
     month_number=1..12;
     day=1..31;
     date=Record
        dyear:year;
        dmonth:month_number;
        dday:day;
       End;
```

Разумеется, тип date можно было определить и так:

```
Type date=Record
         dyear: 1583..3000;
         dmonth: 1..12;
         dday: 1..31;
        End;
```

Остановимся на первом способе, поскольку при решении задач с датами нам потребуется работа и с типами данных year, month_number, day.

С полем записи в программе можно поступать, как с переменной того же типа, что и поле. Для обращения к полю используется составное имя: <имя записи>.<имя поля>.

Пример.

```
Var t:date;
```

Операторы присвоения имеют вид:

```
t.dyear:=1987;
t.dmonth:=12;
t.dday:=30;
```

Обращение к полям записи имеет несколько громоздкий вид. Упрощение этого действия осуществляется с помощью оператора With:

```
With <имя записи> Do <оператор>;
```

Операторы присвоения в этом случае записываются несколько по-другому.

```
With t Do Begin
 dyear:=1987;
 dmonth:=12;
 dday:=30;
End;
```

Оператор выбора (Case)

Оператор выбора Case состоит из выражения (селектора) и списка операторов, каждому из которых предшествует либо одна или несколько констант (называемых константами варианта), либо слово Else. Селектор должен быть порядкового типа, причем порядковые значения верхней и нижней границ этого типа должны находиться в диапазоне от –32 768 до 32 767.

Оператор Case выбирает для выполнения тот оператор, перед которым стоит константа, равная значению селектора или диапазону, содержащему значение селектора. Если в диапазоне выбора не существует такой константы выбора и в операторе имеется часть Else, выполняется оператор, написанный после слова Else. В противном случае — если часть Else отсутствует, выполняется оператор, стоящий за словом End, то есть первый оператор, следующий за оператором Case.

Пример 1.

```
Var c:Char;
Begin
 ReadLn(c)
```

```
 Case c Of
  '0'..'9': WriteLn('c – цифра');
  'a'..'z': WriteLn('c –маленькая латинская буква');
  'A'..'Z': WriteLn('c – большая латинская буква');
  Else WriteLn('c – некоторый другой символ');
 End;
End;
```

Пример 2.

```
Var i:Integer;
Begin
 ReadLn(i);
 Case i Of
  0,2,4,6,8: WriteLn('i – четное число меньше 10.');
  1,3,5,7,9: WriteLn('i – нечетное число меньше 10.');
 End;
End;
```

Экспериментальный раздел занятия

1. Работа с датами.

Используются типы данных year, month_number, day, date, определенные выше. Определим ряд процедур и функций. Год считается високосным (366 дней), если его номер делится без остатка на 4, причем последний год любого столетия считается високосным только в том случае, если его номер делится на 400.

```
Function Leap(x:year):Boolean;
 Begin
  Leap:=(x Mod 400 =0) Or (x Mod 100<>0)
        And (x Mod 4=0);
 End;
```

Задача определения количества дней в месяце решается с помощью следующей функции:

```
Function DayM(x:year;y:month_number):day;
  Begin
   Case y Of
    4,6,9,11:DayM:=30; {В апреле, июне, сентябре
     и ноябре 30 дней.}
    1,3,5,7,8,10,12:DayM:=31; {В январе, марте,
   мае, июле, августе, октябре и декабре 31 день.}
```

```
    2: If Leap(x) Then DayM:=29 Else DayM:=28;
       {Если год високосный, то в феврале 29 дней.}
  End;
 End;
```

Для вывода даты, например в виде 30.12.1987, используется ее преобразование в тип String.

```
Function StringDate(w:Date):String;
 Var s,q:String;
 Begin
  Str(w.dday,s);
  If w.dday<10 Then s:='0'+s; {Преобразуем день
                               даты.}
  s:=s+'.';
  Str(w.dmonth,q); {Преобразуем месяц даты.}
  If w.dmonth<10 Then s:=s+'0'+q+'.'
   Else s:=s+q+'.';
  Str(w.dyear,q); {Преобразуем год даты.}
  s:=s+q;
  StringDate:=s;
 End;
```

Из текущей даты требуется получить дату следующего дня. При решении этой задачи необходимо учитывать, является ли:

- день даты последним днем месяца;
- месяц даты последним месяцем года.

```
Procedure Tomorrow(x:date;Var y:date);
 Begin
  y:=x;
  If x.dday<>DayM(x.dyear,x.dmonth)
   Then y.dday:=x.dday+1 {День даты не является
                          последним днем месяца.}
   Else
    If x.dmonth<>12 Then Begin
          y.dmonth:=x.dmonth+1; {Месяц даты
            не является последним месяцем в году.}
          y.dday:=1;
         End
     Else Begin
          y.dyear:=x.dyear+1;
```

```
            y.dmonth:=1;
            y.dday:=1;
          End;
End;
```

Для определения даты, которая наступит в будущем, через определенное количество дней *x*, можно воспользоваться предыдущей процедурой, продвигаясь вперед на один день *x* раз. Однако шаг продвижения может быть больше, если *w* превышает количество дней в году.

```
Procedure Future(x:Integer;Var y:Date);
{x – количество дней, y – текущая дата.}
 Var i:Integer;
 Function Dyears(z:year):Integer; {Определяем
       количество дней в году.}
  Begin
   If Leap(z) Then Dyears:=366 Else Dyears:=365;
  End;
 Begin
  For i:=1 To y.dmonth-1 Do x:=x+DayM(y.dyear,i);
     {Возвращаемся в начало года, определенного
     текущей датой.}
  x:=x+y.dday; {Прибавляем количество дней из
    даты. Получаем общее количество дней от начала
    текущего года.}
  While x>Dyears(y.dyear) Do Begin {Счет идет по
                                    годам.}
   x:=x-Dyears(y.dyear);
   y.dyear:=y.dyear+1;
  End; {Год будущей даты сформирован.}
  y.dmonth:=1; {Определяем месяц будущей даты.}
  While x>DayM(y.dyear,y.dmonth) Do Begin {Счет
               идет по месяцам.}
   x:=x-DayM(y.dyear,y.dmonth);
   y.dmonth:=y.dmonth+1;
  End; {Месяц будущей даты получен.}
  y.dday:=x; {Оставшиеся дни являются днями
              будущей даты.}
 End;
```

Основная программа для работы с датами может иметь следующий вид:

```
Program My21_1;
 Var t,r:date;
     w:Integer;
{Процедуры и функции.}
 Begin
  WriteLn(' Введите год, месяц, день даты: ');
  ReadLn(t.dyear,t.dmonth,t.dday);
  Tomorrow(t,r);
  WriteLn(StringDate(r));
  WriteLn(' Введите количество дней до будущей
           даты.');
  ReadLn(w);
  Future(w,t);
  WriteLn(StringDate(t));
  ReadLn;
 End.
```

Многочисленные модификации данной программы получаются при выполнении ряда заданий для самостоятельной работы.

2. Начальные сведения из компьютерной геометрии.

Точка на плоскости в декартовой прямоугольной системе координат описывается парой вещественных чисел. Введем соответствующий тип данных — запись. При использовании вещественного типа операции сравнения лучше оформить специальными функциями. Причина понятна: на типе Real в системе программирования Паскаль нет отношения порядка, поэтому записи вида $a=b$, где a и b вещественные числа, лучше не использовать. По этой же причине все вычисления с вещественным типом лучше выполнять с какой-то заранее определенной точностью.

```
Type TPoint=Record
           x, y: Real;
          End;
Const Eps: Real=1E-7; {Точность вычисления.}
      ZeroPnt: Tpoint = (x:0; y:0); {Точка
      с координатами 0,0.}
```

Пример реализации операций сравнений:

```
Function RealEq(a, b: Real): Boolean; {Строго равно.}
 Begin
```

```
  RealEq:=Abs(a-b)<=Eps;
 End;
```

Функция проверки совпадения двух точек на плоскости имеет вид:

```
Function EqPoint(A, B: TPoint):Boolean;
{Совпадают ли точки?}
 Begin
  EqPoint:=RealEq(A.x, B.x) And RealEq(A.y, B.y);
 End;
```

При вычислении расстояния между двумя точками на плоскости удобнее опять же использовать функцию.

```
Function Dist(A, B: TPoint): Real; {Расстояние
между двумя точками на плоскости.}
 Begin
  Dist:=Sqrt(Sqr(A.x-B.x) + Sqr(A.y-B.y));
 End;
```

Любая точка на плоскости определяет вектор ***v*** — направленный отрезок[12], соединяющий точку (0, 0) с точкой (x, y). Вектор ***v*** можно задать в полярной системе координат через его длину (модуль) v и угол α относительно оси OX. Координаты полярной системы координат (v, α) и прямоугольной декартовой (x, y) связаны соотношениями:

$x=v \cdot \cos(\alpha)$, $y=v \cdot \sin(\alpha)$, $v=\sqrt{x^2+y^2}$, $\operatorname{tg}(\alpha)=y/x$.

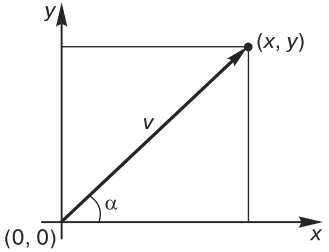


Рис. 1.20. Описание вектора в декартовой и полярной системах координат

Разработаем функцию определения угла α (в радианах) по координатам точки в декартовой системе координат.

[12] Обозначение вектора выделяется полужирным шрифтом.

```
Function GetAngle(w:TPoint):Real; {w точка на
                                   плоскости.}
 Var v,angle:Real;
 Begin
  v:=Dist(w,ZeroPnt); {Расстояние.}
  If RealEq(v,0) Then GetAngle:=0
   Else Begin
     If RealEq(w.x,0) Then
       If w.y>0 Then angle:=Pi/2
        Else angle:=3*Pi/2 {Выделяем особые случаи.
                 Точка находится на одной из осей.}
      Else
       If RealEq(w.y,0) Then
         If w.x>0 Then angle:=0
          Else angle:=Pi
        Else Begin
          angle:=ArcTan(Abs(w.y/w.x));
          {Определяем четверть плоскости.}
          If w.x*w.y>0 Then Begin
             If w.x<0 Then angle:=Pi+angle;
           End
          Else
           If w.x<0 Then angle:=Pi/2+angle
            Else angle:=2*Pi-angle;
        End;
    GetAngle:=angle;
   End;
 End;
```

Как известно, скалярным произведением двух векторов называется число, равное произведению длин этих векторов на косинус угла между ними. Если хоть один из векторов нулевой, то угол не определен, и скалярное произведение по определению считается равным нулю. Таким образом, $(\boldsymbol{v}, \boldsymbol{w}) = v \cdot w \cdot \cos(\alpha)$, где α — угол между векторами ***v*** и ***w***.

Напомним, что косинус угла между векторами ***v*** и ***w*** вычисляется по формуле:

$$\cos(\alpha) = \frac{v.x \cdot w.x + v.y \cdot w.y}{v \cdot w}.$$

Таким образом, скалярное произведение:

$(\boldsymbol{v}, \boldsymbol{w}) = v.x \cdot w.x + v.y \cdot w.y$.

```
Function ScDec(v, w: TPoint): Real;
 {Скалярное произведение векторов в прямоугольной
                    декартовой системе координат.}
 Begin
  ScDec:=v.x*w.x+v.y*w.y;
 End;
```

Вычисление скалярного произведения векторов в полярной системе координат (v, α) и (w, β) выполняется по формуле:

$$\mathbf{v} \cdot \mathbf{w} = v.x \cdot w.x + v.y \cdot w.y =$$
$$= v \cdot \cos(\alpha) \cdot w \cdot \cos(\beta) + v \cdot \sin(\alpha) \cdot w \cdot \sin(\beta) = v \cdot w \cdot \cos(\alpha - \beta).$$

Из этого соотношения следует, что скалярное произведение:

- ненулевых векторов равно нулю тогда и только тогда, когда векторы перпендикулярны;
- больше нуля, если угол между векторами острый, и меньше нуля, если угол тупой.

Из аналитической геометрии известно, что прямая линия на плоскости, проходящая через две точки p_1 и p_2, определяется следующим линейным уравнением от двух переменных:

$$(p_2.x - p_1.x) \cdot (y - p_1.y) = (p_2.y - p_1.y) \cdot (x - p_1.x).$$

После преобразований получаем:

$$-(p_2.y - p_1.y) \cdot x + (p_2.x - p_1.x) \cdot y + (p_2.y - p_1.y) \cdot p_1.x - (p_2.x - p_1.x) \cdot p_1.y = 0,$$

или, после введения соответствующих обозначений,

$$A \cdot x + B \cdot y + C = 0.$$

Если прямые заданы с помощью уравнений $A_1 \cdot x + B_1 \cdot y + C_1 = 0$ и $A_2 \cdot x + B_2 \cdot y + C_2 = 0$, то точка их пересечения, в случае ее существования ($A_1 \cdot B_2 - A_2 \cdot B_1 \neq 0$), определяется по формулам:

$$x = -(C_1 \cdot B_2 - C_2 \cdot B_1)/(A_1 \cdot B_2 - A_2 \cdot B_1),$$
$$y = (A_2 \cdot C_1 - A_1 \cdot C_2)/(A_1 \cdot B_2 - A_2 \cdot B_1).$$

Определим тип данных для описания прямой:

```
Type TLine=Record
           A, B, C: Real;
         End;
```

Процедуру вычисления коэффициентов в уравнении прямой, проходящей через две точки, и функция вычисления координат точки пересечения двух прямых будут иметь вид:

```
Procedure Point2ToLine(A, B: TPoint; Var L: TLine);
 {Определение уравнения прямой по координатам
    двух точек.}
 Begin
  L.A:=B.y-A.y;
  L.B:=A.x-B.x;
  L.C:=-(A.x*L.A + A.y*L.b);
 End;
Function Line2ToPoint(fL, sL: TLine;
                      Var P: TPoint): Boolean;
{Определение координат точки  пересечения двух
    линий. Значение функции равно True, если точка
    есть, и False, если прямые параллельны.}
 Var st: Real;
 Begin
  st:=fL.A*sL.B-sL.A*fL.B;
  If Not(RealEq(st, 0)) Then Begin
    Line2ToPoint:=True;
    P.x:=-(fL.C*sL.B-sL.C*fL.B)/st;
    P.y:=(sL.A*fL.C-fL.A*sL.C)/st;
   End
  Else Line2ToPoint:=False;
 End;
```

Задания для самостоятельной работы

1. Напишите процедуру (или функцию) определения:
   - даты вчерашнего дня;
   - даты, которая была за m дней до указанной даты;
   - количества дней, прошедших от даты t_1 до t_2;
   - дня недели, выпадающего на дату t_1 (для решения задачи необходимо ввести тип данных «день недели» и определенную дату «связать» с конкретным днем недели);
   - годов столетия (1901–2000), которые начинаются и заканчиваются в воскресенье;
   - годов столетия, содержащих максимальное число воскресений.
2. Дана дата вашего рождения (включая и день недели). Найдите несколько дат, когда ваш день рождения снова придется на тот же день недели.

3. Даны даты рождения членов семьи из пяти человек. Столетними юбилеями семьи назовем дни, когда сумма возрастов всех членов семьи кратна 100 (возрасты исчисляются в днях). Найдите такие даты.
4. Дано время суток, описанное следующим образом:

```
Type  time = Record
             h: 0..23;
             m, s: 0..59
            End;
```

Напишите:

- логическую функцию для проверки, предшествует ли время t_1 времени t_2 (в рамках суток);
- процедуру, присваивающую параметру t_1 время, на 1 секунду большее времени t (учесть возможную смену суток).

5. Дано следующее описание данных:

```
Const n=300;
Type MyRecord = Record
                 key: Integer; {Ключ.}
                 name : String;
                End;
Table=Array[1..n] Of  MyRecord.
```

Считая, что записи в массиве имеют различные ключи, напишите:

- процедуру, упорядочивающую записи массива по убыванию значений поля *key*;
- логическую функцию Search(t, k, h), определяющую, есть ли в массиве t (все записи которой уже упорядочены по возрастанию значений поля *key*) запись со значением поля *key*, равным k, и, если есть, присваивающую ее номер параметру h.

6. Дан массив, содержащий информацию об учениках некоторой школы (данные вводятся из файла). Напишите программы:

- формирования второго массива с данными об учениках только девятых классов;
- определения количества учеников восьмых и девятых классов.

7. Багаж пассажира характеризуется количеством вещей и общим весом вещей. Дан массив, содержащий сведения о багаже нескольких пассажиров (данные вводятся из файла). Сведения о багаже каждого пассажира представляют собой запись с двумя полями: одно поле целого типа (количество вещей) и одно — действительное (вес в килограммах). Определите:
   - багаж, в котором средний вес одной вещи отличается не более чем на 0,3 кг от общего среднего веса одной вещи;
   - число пассажиров, имеющих более двух вещей, и число пассажиров, количество вещей которых превосходит среднее число вещей;
   - имеется ли пассажир, багаж которого состоит из одной вещи весом менее 30 кг.
8. Разработайте процедуры перевода координат точки из декартовой в полярную (и обратно) системы координат. Для описания точки в полярной системе координат использовать тип данных:

```
Type  TPol=Record
                  r,ang:Real;
          End;
```

Описание процедур должно иметь следующий вид:

```
Procedure TurnDecPol(w: TPoint; Var q: TPol);
{Перевод из декартовой системы координат
   в полярную.}
Procedure TurnPolDec(q: TPol; Var w: TPoint);
{Перевод из полярной системы координат
     в декартову.}
```

9. Разработайтеь процедуры сложения и вычитания векторов в декартовой и полярной системах координат.

Пример:

```
Procedure AddPol(a, b: TPol; Var c: TPol); {Сумма
     двух векторов в полярной системе координат.}
 Begin
  c.ang:=(a.ang+b.ang)/2;
  c.r:=Sqrt(Sqr(a.r*Cos(a.ang)-b.r*Cos(b.ang))+
          Sqr(a.r*Sin(a.ang)-b.r*Sin(b.ang)));
 End;
```

10. Координаты точек отрезка (p_1, p_2) можно задать двумя параметрическими уравнениями от одной независимой переменной t:

$p.x = p_1.x + (p_2.x - p_1.x) \cdot t$;
$p.y = p_1.y + (p_2.y - p_1.x) \cdot t$.

При $0 \le t \le 1$ точка $p(x, y)$ лежит на отрезке, а при $t < 0$ или $t > 1$ — вне отрезка на прямой линии, продолжающей отрезок.

Даны координаты двух точек на плоскости, определяющих отрезок, и координаты третьей точки. Определите, принадлежит ли третья точка отрезку?

11. Даны два отрезка. На рис. 1.21 точка пересечения отрезков, если она существует, обозначена как $p=(x, y)$. Первый отрезок задан точками $p_1=(x_1, y_1)$ и $p_2=(x_2, y_2)$, а второй — точками $p_3=(x_3, y_3)$ и $p_4=(x_4, y_4)$. Определите координаты точки пересечения отрезков.

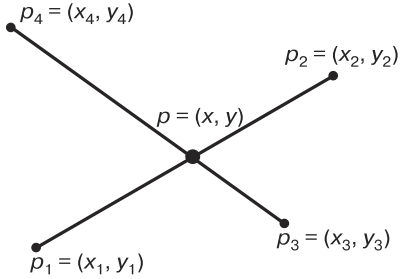


Рис. 1.21. Пример пересечения отрезков

12. Дано n точек. Найдите пару точек с максимальным расстоянием между ними.

13. Дано множество прямых. Подсчитайте количество точек пересечения этих прямых.

14. Дано два множества точек. Найдите пересечение и разность этих множеств.

15. Известно, что два отрезка находятся на одной прямой. Определите их взаимное расположение.

16. Дано множество точек. Найдите в этом множестве две точки, такие, чтобы количество точек, лежащих по разные

стороны от прямой, проходящей через эти две точки, различалось наименьшим образом.

17. Даны координаты вершин треугольника и точки. Определите взаимное расположение этих геометрических объектов.

18. Даны координаты вершин треугольника и концов отрезка. Определите их взаимное расположение.

19. Даны координаты вершин двух треугольников. Определите их взаимное расположение.

20. Дано множество прямых. Определите количество частей, на которые они разбивают плоскость.

Примечания

1. В задачах все геометрические объекты задаются на плоскости.
2. Исходные данные в задачах вводятся из файла.

Часть II

Фундаментальные алгоритмы

Занятие № 22. Поиск данных

План занятия

1. Линейный поиск.
2. Бинарный поиск.
3. Случайный поиск.
4. Поиск элемента произвольного массива за линейное время.
5. Поиск второго минимального элемента в массиве.
6. Выполнение самостоятельной работы.

Линейный поиск

Основной вопрос задачи поиска — нахождение в заданной совокупности данных элемента, обладающего определенным свойством. Большинство таких задач поиска сводится к простейшей — к поиску в массиве элемента с заданным значением.

Пусть требуется найти в массиве A элемент x. В данном случае известно только значение разыскиваемого элемента, никакой дополнительной информации о нем или о массиве, в котором его надо искать, нет. Начнем последовательный просмотр массива и сравнение значения очередного рассматриваемого элемента массива A с x. Напишем фрагмент:

```
For i:=1 To N Do
 If A[i]= x  Then k:=i;
```

Так находится место (номер) последнего элемента в A, равного x, при этом массив просматривается полностью. А если требуется найти номер первого такого элемента?

```
For i:=N DownTo 1 Do
 If A[i]= x Then k:=i;
```

А если исключить лишние проверки? Совпадение найдено, зачем продолжать работу?

```
i:=1;
While (i<=N) And (A[i]x) Do i:=i+1;
If i=N+1 Then <x нет в A>
 Else <значение i указывает на место совпадения>;
```

С помощью данной логики мы решаем задачу поиска первого совпадения. Для нахождения последнего совпадения ее требуется изменить.

```
i:=N;
While (i>0) And (A[i]<>x) Do i:=i-1;
If i=0 Then <x нет в A>
 Else <значение i указывает на место совпадения>;
```

Примечания

1. Необходимо обратить внимание на порядок проверки составного условия цикла. Предполагается, что второе условие не проверяется, если результат логического выражения ясен после проверки первого условия. В противном случае возникает ошибка из-за выхода индекса за границы массива.

2. При изучении данной темы уместно использование директивы {$R+}.

3. В качестве упражнения можно использовать замену цикла While на цикл Repeat – Until.

Использование техники «барьерных» элементов упрощает запись условия завершения цикла, но требует изменения описания типа массива *A* (введения дополнительного элемента).

```
Type MyArray=Array[0..NMax] Of Integer;
```

или

```
Type MyArray=Array[1..NMax+1] Of Integer;
Var A:MyArray;
```

Значение *Nmax* определяется в разделе констант.

```
i:=1;
A[N+1]:=x;
While A[i]<>x Do i:=i+1;
If i=N+1 Then <x нет в A>
 Else <значение i указывает на место совпадения>;
```

Или для поиска последнего совпадения:

```
i:=N;
A[0]:=x;
While A[i]<>x Do i:=i-1;
If i=0 Then <x нет в A>
 Else <значение i указывает на место совпадения>;
```

Очевидно, что число сравнений зависит от места нахождения искомого элемента. В среднем количество сравнений равно (N+1) Div 2, в худшем случае, если элемента нет в массиве, — N сравнений. Эффективность поиска — O(N).

Бинарный поиск

Пусть известна некоторая информация о данных, среди которых ведется поиск, например, что элементы массива расположены в неубывающем порядке. В этом случае удается уменьшить время работы по нахождению элемента, используя бинарный поиск, который еще называют двоичным, дихотомическим, методом деления пополам и др.

Итак, требуется определить место x в отсортированном (например, в порядке неубывания) массиве A. Идея бинарного метода заключается в разделении массива пополам, сравнении элемента x с элементом, находящимся на границе разделяемых половин, и выборе по результатам сравнения одной из половин для дальнейшего поиска. И так до тех пор, пока не будет найден искомый элемент или не будет проанализирован весь массив.

Пример.

Пусть x=6, а массив A состоит из 10 элементов:
3 5 6 8 12 15 17 18 20 25.

1-й шаг. Найдем номер среднего элемента: $m = \left[\frac{1+10}{2}\right] = 5$.

Так как 6<A[5], то далее можем рассматривать только элементы, индексы которых меньше 5:
3 5 6 8 ~~12 15 17 18 20 25~~.

2-й шаг. Рассматривая лишь первые 4 элемента массива, находим индекс среднего элемента этой части $m = \left[\frac{1+4}{2}\right] = 2$,

6>A[2], следовательно, первый и второй элементы из рассмотрения исключаются:

~~3~~ ~~5~~ 6 8 ~~12 15 17 18 20 25~~.

3-й шаг. Рассматриваем два элемента, значение $m = \left[\frac{3+4}{2}\right] = 3$:

~~3 5~~ ~~6~~ 8 ~~12 15 17 18 20 25~~;

A[3]=6. Элемент найден, его номер — 3.

Программная реализация бинарного поиска может иметь следующий вид.

```
Procedure Search;
 Var i,j,m:Integer;
     f:Boolean;
 Begin
  i:=1;
  j:=N;{На первом шаге рассматривается весь массив.}
  f:=False;{Признак того, что X не найден.}
  While (i<=j) And Not f Do Begin
   m:=(i+j) Div 2;
   {Или m:=i+(j-i) Div 2;, так как i+(j-i) Div 2=
   (2*i+(j-i)) Div 2=(i+j) Div 2.}
   If A[m]=x Then f:=True {Элемент найден. Поиск
                           прекращается.}
    Else
     If A[m]<x Then i:=m+1
        {Исключаем из рассмотрения левую часть A.}
      Else j:=m-1;
            {Правую часть.}
 End;
End;
```

Рассмотрим еще одну реализацию изучаемого алгоритма поиска, рекурсивную. Значение переменной *t* является индексом искомого элемента или нулевым значением, если элемент не найден.

```
Procedure Search(i,j:Integer;Var t:Integer);
{Массив A и переменная x - глобальные величины.}
 Var m:Integer;
 Begin
  If i>j Then t:=0
   Else Begin
        m:=(i+j) Div 2;
```

```
        If A[m]<x Then Search(m+1,j,t)
         Else
          If A[m]>x Then Search(i,m-1,t)
           Else t:=m;
       End;
End;
```

Поскольку каждое сравнение уменьшает диапазон поиска приблизительно в два раза, общее количество сравнений имеет порядок $O(\log_2 N)$.

Случайный поиск

Пусть необходимо найти k-й по порядку элемент в неупорядоченном массиве A. Идею бинарного поиска можно использовать и в данном случае следующим образом:

1. Выбирается случайным образом элемент с номером q.

2. Массив A разбивается на три части: элементы, меньшие $A[q]$, равные $A[q]$ и большие $A[q]$.

3. В зависимости от количества элементов в каждой части выбирается одна из частей для дальнейшего поиска.

Теоретическая оценка числа сравнений имеет порядок $O(k \cdot N)$, т. е. для худшего случая $O(N^2)$, но на практике поиск работает значительно быстрее.

```
Function Search(N,k:Integer;A:MyArray):Integer;
{N - количество элементов в массиве A, k - номер
          по порядку разыскиваемого элемента.}
 Var i,cnl,cne,cnm,q:Integer;
     Sl,Se,Sm:MyArray;
 Begin
  If N=1 Then Search:=A[1]
   Else Begin
    FillChar(Sl,SizeOf(Sl),0); {Элементам
                массивов Sl, Se, Sm присваиваются
                нулевые значения.}
    FillChar(Se,SizeOf(Se),0);
    FillChar(Sm,SizeOf(Sm),0);
    cnl:=0;
    cne:=0;
    cnm:=0;
    q:=Integer(Random(N))+1;
    For i:=1 To N Do{Разбиение на три части.}
```

```
    If A[i]<A[q] Then Begin
                    cnl:=cnl+1;
                    Sl[cnl]:=A[i];
                   End
     Else
      If A[i]=A[q] Then Begin
                      cne:=cne+1;
                      Se[cne]:=A[i];
                     End
       Else Begin
            cnm:=cnm+1;
            Sm[cnm]:=A[i];
           End;
   {Выбор части для дальнейшего поиска.}
    If cnl>=k Then Search:=Search(cnl,k,Sl)
     Else
      If cnl+cne>=k Then Search:=A[q]
       Else Search:=Search(cnm,k-cnl-cne,Sm);
 End;
End;
```

Поиск элемента массива за линейное время

Продолжим рассмотрение задачи поиска k-го по порядку элемента в неупорядоченном массиве A. Для реализации этой схемы поиска требуется уметь сортировать массив из 5 элементов за 7 сравнений (алгорим будет приведен в материале следующего занятия). Пусть он нам известен. Обозначим его как Sort(N,A), где N — количество элементов в массиве A. Идея алгоритма базируется на бинарном поиске. Поясним суть метода с помощью рис. 2.1. Из отсортированных пятиэлементных массивов формируется массив M из их средних элементов —

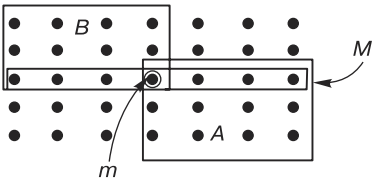


Рис. 2.1. Один шаг схемы поиска элемента в неупорядоченном массиве

медиан. В массиве M выбирается средний элемент m и относительно него исходный массив разбивается на три части, как в предыдущей схеме случайного поиска. В части A на рис. 2.1 выделены элементы, заведомо большие m, в части B — меньшие m. За счет однократного применения действий этой схемы как минимум четвертая часть элементов исходного массива исключается из дальнейшего поиска.

```
Function Search(N,k:Integer;A:MyArray):Integer;
{N - количество элементов в массиве A, k - номер
    элемента (k≤N) по порядку, который требуется
    найти.}
 Var i,j,cnl,cne,cnm,q,r:Integer;
    M,Sl,Se,Sm,Y:MyArray;
 Begin
  If N<=5 Then Begin {Если количество элементов
       в массиве меньше или равно 5, то сортируем
       элементы массива за 7 сравнений и выбираем
       k-й элемент.}
                Sort(N,A);
                Search:=A[k];
               End
   Else Begin
        i:=1;
        r:=0;
        While i<=N Do Begin
          j:=0;
          While (j<=5) And (i+j<=N) Do Begin
    {Разбиваем на пятиэлементные массивы
    (не более).}
           j:=j+1;
           Y[j]:=A[i+j-1];
          End;
         i:=i+5;
         Sort(j,Y);
         r:=r+1;
         M[r]:=Y[j Div 2+1]; {Средний элемент
                    отсортированного массива Y
                    записываем в массив M.}
        End;
       q:=Search(r,r Div 2,M);{Находим средний
          по значению элемент в массиве медиан.}
```

```
    cnl:=0;
    cne:=0;
    cnm:=0;
    FillChar(Sl,SizeOf(Sl),0);
    FillChar(Se,SizeOf(Se),0);
    FillChar(Sm,SizeOf(Sm),0);
    For i:=1 To N Do {Разбиваем массив на три
                      части.}
     If A[i]<q Then Begin
                  cnl:=cnl+1;
                  Sl[cnl]:=A[i];
                 End
      Else
       If A[i]=q Then Begin
                    cne:=cne+1;
                    Se[cne]:=A[i];
                   End
        Else Begin
             cnm:=cnm+1;
             Sm[cnm]:=A[i];
            End;
   {Выбор части массива для дальнейшего поиска.}
     If cnl>=k Then Search:=Search(cnl,k,Sl)
      Else
       If cnl+cne>=k Then Search:=A[q]
        Else Search:=Search(cnm,k-cnl-cne,Sm);
   End;
End;
```

Оценим T(N), где T — время работы алгоритма.

После поиска элемента m не менее половины найденных медиан будут больше или равны m. Следовательно, по крайней мере половина $\lceil N/5 \rceil$[13] групп дадут по три числа, больших или равных m, кроме, может быть, последней группы и группы, содержащей m. Итак, $3 \cdot \left(\left\lceil \frac{1}{2} \cdot \left\lceil \frac{N}{5} \right\rceil \right\rceil - 2 \right) \geq 3 \cdot \frac{N}{10} - 6$ элементов исключаются из дальнейшей обработки. Остается $7 \cdot \frac{N}{10} + 6$ элементов.

[13] Округления числа x до ближайшего целого в меньшую и большую стороны называют «пол» и «потолок» x и обозначают $\lfloor x \rfloor$ и $\lceil x \rceil$ соответственно.

Поиск среднего элемента в массиве медиан требует $T\left(\frac{N}{5}\right)$ времени. Рекурсивный вызов процедуры поиска при разбивке массива требует $T\left(7\cdot\frac{N}{10}+6\right)$. Остальные шаги алгоритма, включая сортировку 5-элементных массивов, выполняются за время $O(N)$. Таким образом, $T(N)\le T\left(\frac{N}{5}\right)+T\left(7\cdot\frac{N}{10}+6\right)+O(N)$. Сумма коэффициентов при N в правой части: $\frac{1}{5}+\frac{7}{10}=\frac{9}{10}$ меньше 1. Известно, что в этом случае $T(N)\le c\cdot N$ для некоторой константы c. Следовательно, временна́я сложность алгоритма есть $O(N)$.

Примечание

В разных источниках приводятся различные схемы оценки времени работы этого алгоритма и значения N (50 или 70), начиная с которых время поиска пропорционально $O(N)$.

Поиск второго минимального элемента в массиве

Известно, что второй минимальный элемент в массиве из N элементов можно найти за $N+\lceil\log_2 N\rceil-2$ сравнений. Так, при N=8 это 9 сравнений. Попробуйте свои силы: не читая дальнейшее изложение, самостоятельно найдите алгоритм решения задачи.

Очевидно, что без поиска минимального элемента найти второй минимальный невозможно. Поиск минимального элемента требует $N-1$ сравнений и от этого никуда не уйти. Осталось $\lceil\log_2 N\rceil-1$ сравнение. Если искать по прежней схеме, исключив найденный элемент, то потребуется еще $N-2$ сравнения, а в сумме $2\cdot N-3$ сравнения. При этом мы как бы забываем то, что делали на первом шаге. Единственно возможный (логически) выход заключается в том, чтобы создавать некую дополнительную информацию на первом шаге и использовать ее для поиска второго минимального элемента. Но как это сделать?

Пусть N=8 и элементы массива A имеют значения: 81, 34, 2, 90, 51, 45, 14, 31. На рис. 2.2 изображена схема поиска минимального элемента: сравниваем пары соседних элементов, затем — минимальные элементы пар и так далее.

Если убрать записи в круглых скобках на рис. 2.2, то это обычный поиск минимального элемента в массиве за $N-1$ сравнение (напишите процедуру такого поиска).

А сейчас обратимся к записям в круглых скобках. Рассмотрим, например, 2(34,90). Почему мы не включили 81? Причина в том, что этот элемент не может быть вторым минимальным элементом. Только 34 и 90 кандидаты на это место. Точно так же и 14(45, 31). Элемент 51 исключается из «цепочки» претендентов. Последнее 7-е сравнение оставляет в этой «цепочке» 2(14, 34, 90) только три элемента, ибо 45 и 31 не могут претендовать на эту роль, 14 заведомо меньше их, а этот элемент включен в «цепочку». Итак, в списке 3 элемента, за 2 сравнения мы находим второй минимальный элемент, общее количество сравнений равно 9.

Постройте точно такую же схему для массива из 16 элементов. Через 15 сравнений в последней «цепочке» окажется 4 элемента. За 3 сравнения мы найдем второй минимальный элемент. Итого, 18 сравнений.

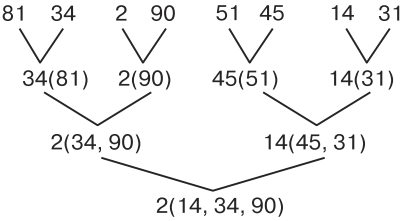


Рис. 2.2. Схема поиска второго минимального элемента

Перейдем к реализации данного алгоритма. Описание данных программной реализации имеет вид:

```
Program SearchTwoMin;{Без ввода исходных данных и
                                вывода результата.}
 Const MaxN=10000;
 Type MyArray=Array[1..MaxN] Of Integer;
 Var A, iNext, iFirst: MyArray; {Массив элементов.
                Номер следующего элемента цепочки.
                Номера первых элементов цепочек.}
   N: Integer;{Количество элементов в массиве A.}
   cnt: LongInt;{Счетчик числа сравнений.}
   res: Integer;{Второй минимальный элемент.}
```

Предположим, что после поиска минимального элемента в дополнительном массиве *iNext* записаны индексы (номера)

элементов последней цепочки, например, так: *, 4, 7, 0, *, *, 2, *, а индекс минимального элемента массива (для рассматриваемого примера это 3) является значением переменной *iFirst*[1]. Символом «*» обозначено не интересующее нас значение, оно может быть любым, а 0 — признак конца «цепочки». Задача решена. Просмотр элементов последней «цепочки» по этим данным реализуется с помощью следующей процедуры.

```
Procedure Search;
  Var t: Integer;
   Begin
     t:=iNext[iFirst[1]];{Для примера значением t
              является 7. Определяем номер первого
              элемента цепочки.}
     res:=A[t];{Начальное присвоение.}
     t:=iNext[t]; {Переход ко второму элементу
              «цепочки».}
     While t<>0 Do Begin
      cnt:=cnt+1; {Изменяем значение счетчика
              числа сравнений.}
      If A[t]<res Then res:=A[t]; {Изменяем
              значение минимального элемента.}
      t:=iNext[t]; {Переадресация к следующему
              элементу «цепочки».}
     End;
   End;
```

Осталось получить значения элементов массивов *iNext* и *iFirst* в процессе поиска минимального элемента массива *A* (напомним его значения для рассматриваемого примера: 81, 34, 2, 90, 51, 45, 14, 31). В начальный момент времени у нас 8 цепочек по одному элементу. После первого шага работы получаются 4 цепочки по два элемента. Номер первого элемента первой цепочки 2 (*iFirst*[1]), номер следующего элемента 1 (*iNext*[2]). Номер первого элемента второй цепочки 3 (*iFirst*[2]), номер следующего элемента 4 (*iNext*[3]). Номер первого элемента третьей цепочки 6 (*iFirst*[3]), номер следующего элемента 5 (*iNext*[6]). Номер первого элемента четвертой цепочки 7 (*iFirst*[4]), номер следующего элемента 8 (*iNext*[7]).

Результаты действий на первом шаге обработки приведены в табл. 2.1.

Таблица 2.1

	iFirst								*iNext*							
Начальные значения	1	2	3	4	5	6	7	8	0	0	0	0	0	0	0	0
A[iFirst[1]]>*A*[iFirst[2]]	2	2	3	4	5	6	7	8	0	1	0	0	0	0	0	0
A[iFirst[3]]<*A*[iFirst[4]]	2	3	3	4	5	6	7	8	0	1	4	0	0	0	0	0
A[iFirst[5]]>*A*[iFirst[6]]	2	3	6	4	5	6	7	8	0	1	4	0	0	5	0	0
A[iFirst[7]]<*A*[iFirst[8]]	2	3	6	7	*	*	*	*	0	1	4	0	0	5	8	0

В результате осталось 4 элемента, точнее, 4 цепочки элементов, номера их первых элементов являются значениями *iFirst*[1] – *iFirst*[4]. Символом «*» в табл. 2.1 отмечены неиспользуемые элементы массива *iFirst*. После второго шага остается две цепочки элементов (табл. 2.2).

Таблица 2.2

	iFirst				*iNext*							
Начальные значения	2	3	6	7	0	1	4	0	0	5	8	0
A[iFirst[1]]>*A*[iFirst[2]]	3	3	6	7	0	4	2	0	0	5	8	0
A[iFirst[3]]>*A*[iFirst[4]]	3	7	*	*	0	4	2	0	0	8	6	0

Номер первого элемента первой цепочки — 3 (*iFirst*[1]), затем 2 (*iNext*[3]) и 4 (*iNext*[2]). Номер первого элемента второй цепочки — 7 (*iFirst*[2]), затем 6 (*iNext*[7]) и 8 (*iNext*[6]). Осталось последнее сравнение элементов *A*[*iFirst*[1]] и *A*[*iFirst*[2]]. Элементы массива *IndNext* после корректировки ссылок примут значения: 0, 4, 7, 0, 0, 8, 2, 0, а *IndFirst*[1] останется равным 3. Значение *A*[7] больше *A*[3], про цепочку седьмого элемента забываем, его включаем первым элементом в цепочку третьего элемента, а старое значение *iNext*[3] становится значением *iNext*[7].

```
Procedure Create;
 Var i,lf,rg,t: Integer;
 Begin
  cnt:=0;{Счетчик числа сравнений.}
  FillChar(iNext, SizeOf(iNext), 0);{Начальные
                значения элементов массива iNext.}
  For i:=1 To N Do iFirst[i]:=i;{Начальные
                значения элементов массива iFirst.}
  t:=N;
```

```
  While t>1 Do Begin {Номер шага, t Div 2 -
                      количество сравнений на данном шаге.
                      Для примера: 8, 4, 2.}
   i:=2;
   While i<=t Do Begin
     cnt:=cnt+1;{Изменяем значение счетчика числа
                 сравнений.}
     lf:=i-1;
     rg:=i-1;
     If A[iFirst[i-1]]<A[iFirst[i]] Then rg:=i
      Else lf:=i; {В сравниваемой паре элементов
        массива A: rg - номер большего, а lf - номер
        меньшего элементов, определяемых по
        значению из IndFirst.}
{Следующие три строки кода - самый сложный момент
логики. Связываем цепочки сравниваемых элементов
массива A.  Изменяем номер первого элемента новой
цепочки.}
     iNext[iFirst[rg]]:=iNext[iFirst[lf]];
     iNext[iFirst[lf]]:=iFirst[rg];
     iFirst[i Div 2]:=iFirst[lf];
     i:=i+2;
   End;
   {Учитываем нечетность значения N.}
   If t Mod 2=1 Then
      iFirst[(t+1) Div 2]:=iFirst[t];
   t:=(t+1) Div 2;
   End;
  End;
```

Основная программа, как обычно, состоит из вызова процедур:

```
Begin
 Init;{Ввод исходных данных.}
 Create;
 Search;
 Done;{Вывод результата.}
End.
```

Задания для самостоятельной работы

1. Во входном файле дан массив A и массив из элементов, поиск которых будет осуществляться в массиве A. Измените массив A таким образом, чтобы суммарное количество сравнений при поиске элементов было минимальным.

   Примечание

   По данным из второго массива формируются частоты поиска каждого элемента, и затем элементы массива A сортируются в порядке убывания значений этих частот («метод перемещения в начало»).

2. Дан массив A и число x.

   а) Переставьте элементы массива таким образом, чтобы вначале были записаны элементы, меньшие или равные x, а затем — большие или равные x.

   б) Переставьте элементы массива таким образом, чтобы вначале были записаны элементы, меньшие x, затем равные x и, наконец, большие x.

   Примечание

   Указанные перестановки следует сделать за один просмотр A и без использования дополнительных массивов.

3. Ниже приведены три версии процедуры бинарного поиска. Какая из них верная? Исправьте ошибки. Какая из них более эффективная?

   а)

```
Procedure Search;
 Var i, j, k: Integer;
 Begin
  i:=1;j:=N;
  Repeat
   k:=(i+j) Div 2;
   If x<=A[k] Then j:=k-1;
   If A[k]<=x Then i:=k+1;
  Until i>j;
 End;
```

   б)

```
Procedure Search;
 Var i, j, k: Integer;
 Begin
  i:=1; j:=N;
  Repeat
```

```
    k:=(i+j)Div 2;
    If x<A[k] Then j:=k Else i:=k+1;
   Until i>=j;
  End;
```

в)

```
Procedure Search;
 Var i, j, k: Integer;
 Begin
  i:=1; j:=N;
  Repeat
   k:=(i+j) Div 2
   If A[k]<x Then i:=k Else j:=k;
  Until (A[k]=x) Or (i>=j);
 End;
```

5. Дан массив из N различных целых чисел. Разрешенными операциями являются сложение двух элементов массива и сравнение сумм. Найдите наибольший элемент массива за минимальное количество сравнений.
6. Пусть $X[1..N]$ и $Y[1..N]$ — два возрастающих массива из различных элементов. Найдите за время $O(\log_2 N)$ средний элемент множества, полученного объединением этих массивов.
7. Дана позиция элемента *Pos* в неупорядоченном массиве A из N элементов и количество допустимых сравнений L. Требуется определить все интервалы значений N в диапазоне от 1 до 10000, при которых элемент A с номером *Pos* может быть найден за L сравнений с помощью алгоритма бинарного поиска.

Разберите и проверьте правильность приведенного решения.

```
Function Can(N: Integer): Boolean;
{Модифицированная схема бинарного поиска.}
 Var i,p,q,ll: Integer;
 Begin
  p :=0;
  q :=N-1;
  ll :=0;
  While p <= q Do  Begin
    i := (p+q) ShR 1;
```

```
    ll:=ll+1;
    If i = Pos Then  Begin
       Can := (ll=L);
       Exit;
     End
    Else
     If i > Pos Then q := i-1 Else p := i+1;
    End;
    Can := False;
  End;
Procedure Solve;
  Const MaxN = 10000;
        MaxCnt = MaxN ShR 1;
  Var i,j: Integer;
       Res: Array [1..MaxCnt,1..2] Of Integer;
       Pos, L, cnt: Integer;
  Begin
   ReadLn(Pos, L);
   cnt := 0;
   i := Pos;
   Repeat
     j := i;
     While (j <= MaxN) And Can(j) Do j:=j+1;
     If i < j Then Begin {Запоминаем интервал
значений.}
       cnt:=cnt+1;
       Res[cnt,1] := i;
       Res[cnt,2] := j-1;
     End;
     i := j+1;
   Until (i >= MaxN);
   WriteLn(cnt);
   For i := 1 To cnt Do
    WriteLn( Res[i,1],' ',Res[i,2]);
  End;
```

Занятие № 23. Алгоритмы сортировки с временной сложностью O(n^2)

План занятия

1. Постановка задачи сортировки данных.
2. Сортировка методом простого выбора.
3. Сортировка методом простого обмена.
4. Сортировка методом простых вставок.
5. Эксперименты с программами.
6. Выполнение самостоятельной работы.

Постановка задачи сортировки данных

Для решения многих задач необходимо упорядочивать исходные данные (множество объектов) по определенному признаку. Процесс такого упорядочения называется *сортировкой*.

Для простоты изложения мы будем работать с одномерным массивом из целых чисел

```
Type MyArray=Array[1..NMax] Of Integer;
Var A:MyArray;
```

где значение $Nmax$ определяется в разделе констант.

Алгоритмы сортировки отличаются друг от друга степенью эффективности, под которой понимается количество сравнений и количество обменов, произведенных в процессе работы. Как правило, мы будем оценивать эффективность количеством операций сравнения (порядком этого значения). Заметим, что элементы массива можно сортировать:

- по возрастанию — каждый следующий элемент больше предыдущего: $A[1] < A[2] < \ldots < A[N]$;
- по неубыванию — каждый следующий элемент не меньше предыдущего, то есть больше или равен ему: $A[1] \le A[2] \le \ldots \le A[N]$;
- по убыванию — каждый следующий элемент меньше предыдущего: $A[1] > A[2] > \ldots > A[N]$;
- по невозрастанию — каждый следующий элемент не больше предыдущего, то есть меньше или равен ему: $A[1] \ge A[2] \ge \ldots \ge A[N]$.

Научившись выполнять одну сортировку, не составит особого труда изменить ее, чтобы получить другую.

Сортировка методом простого выбора

Рассмотрим идею этого метода на примере.
Пусть исходный массив *A* состоит из 10 элементов:
5 13 7 9 1 8 16 4 10 2.
После сортировки массив должен выглядеть так:
1 2 4 5 7 8 9 11 13 16.
Введем условные обозначения:

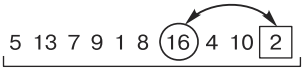
(16) — максимальный элемент текущей части массива;

[7] — элемент, с которым производится обмен;

40 15 32 4 — часть массива, которая рассматривается на данном этапе сортировки;

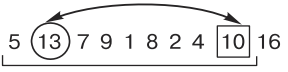
 — операция обмена элементов.

Тогда процесс сортировки можно представить так:
1-й шаг. Рассмотрим весь массив и найдем в нем максимальный элемент — 16. Он находится на седьмом месте. Поменяем его местами с последним элементом — с числом 2.



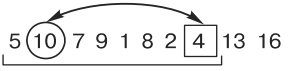
Максимальный элемент записан на свое место.

2-й шаг. Рассмотрим часть массива, исключая последний элемент. Максимальный элемент этой части — 13, стоящий на втором месте. Поменяем его местами с последним элементом этой части — с числом 10.



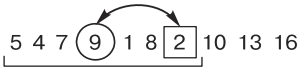
Отсортированная часть массива состоит из двух последних элементов.

3-й шаг. Снова уменьшим рассматриваемую часть массива на один элемент. В данном случае следует поменять местами второй элемент (его значение — 10) и последний элемент этой части — число 4.



Отсортированная часть массива состоит из 3 элементов.

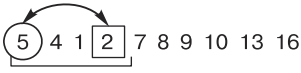
4-й шаг.



5-й шаг. Максимальный элемент этой части массива является последним в ней, поэтому его нужно оставить на старом месте.

6-й шаг.

7-й шаг.



8-й шаг.

9-й шаг.

Приведем фрагмент процедуры сортировки элементов массива методом простого выбора.

```
Procedure Sort(N:Integer; Var A:MyArray);
 Var i, j, k: Integer;
    m:Integer;{Значение максимального элемента
              рассматриваемой части массива.}
 Begin
{Цикл по длине рассматриваемой части массива.}
  For i:=N DownTo 2 Do Begin
{Поиск максимального элемента и его номера
 в текущей части массива.}
    k:=i;
```

```
    m:=A[i];{Начальные значения максимального
    элемента и его индекса в рассматриваемой части
    массива.}
    For j:=1 To i-1 Do
     If A[j]>m Then Begin k:=j; m:=A[j] End;
     If k<>i Then Begin{Перестановка элементов.}
                 A[k]:=A[i];
                 A[i]:=m;
                End;
  End;
 End;
```

Оценим количество сравнений c. При первом просмотре $c=N-1$, при втором — $N-2$, при последнем — 1.

Общее количество $c=N-1+N-2+...+1=N\cdot(N-1)/2$ (сумма элементов арифметической прогрессии) или $c=O(N^2)$. Последняя запись означает, что с ростом значения N количество сравнений будет расти как квадратичная функция от N.

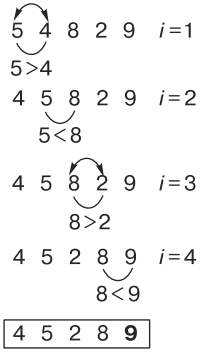
Сортировка методом простого обмена

Рассмотрим идею метода на примере.

Отсортируем по возрастанию массив из 5 элементов: 5 4 8 2 9.

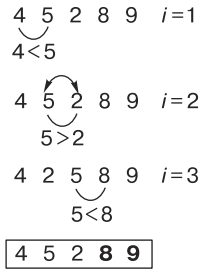
Определим i как номер шага при просмотре массива слева направо.

Первый просмотр: рассматривается весь массив.



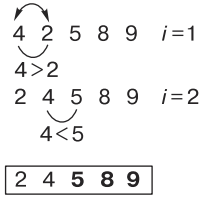
Таким образом, максимальный элемент 9 оказывается на своем месте.

Второй просмотр: рассматриваем часть массива с первого до предпоследнего элемента.



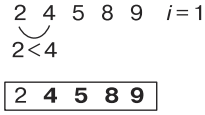
Таким образом, элемент 8 — на своем месте.

Третий просмотр: рассматриваемая часть массива содержит три первых элемента.



5 — на своем месте.

Четвертый просмотр: рассматриваем последнюю пару элементов.



4 — на своем месте.

Наименьший элемент 2 оказывается на первом месте. Заметим, что количество просмотров элементов массива равно $N-1$.

Итак, наш массив отсортирован. Этот метод также называют методом «пузырька». Его называют так потому, что в процессе выполнения сортировки более «легкие» элементы (элементы с заданным свойством, в нашем случае максимальные) мало-помалу всплывают на «поверхность» (оказываются на своем месте).

```
Procedure Sort(N:Integer;Var A:MyArray);
 Var k,i,w:Integer;{k - номер просмотра,
        изменяется от 1 до N-1; i - номер первого
        элемента рассматриваемой пары; w -
        рабочая переменная для перестановки
        местами элементов массива.}
 Begin
  For k:=1 To N-1 Do{Цикл по номеру просмотра.}
   For i:=1 To N-k Do
    If A[i]>A[i+1] Then {Перестановка элементов.}
     Begin
      w:=A[i];
      A[i]:=A[i+1];
      A[i+1]:=w;
     End;
 End;
```

При сортировке методом пузырька выполняется $N-1$ просмотров, на каждом i-м просмотре производится $N-i$ сравнений. Общее количество сравнений $c=N \cdot (N-1)/2$, т. е. порядок $O(N^2)$.

Сортировка методом простых вставок

Сортировка этим методом производится последовательно шаг за шагом. На i-м шаге считается, что часть массива, содержащая первые $i-1$ элементов, уже упорядочена, т. е. $A[1] \le A[2] \le \ldots \le A[i-1]$. Далее берется i-й элемент, и для него подбирается место в отсортированной части массива, такое, что после его вставки упорядоченность не нарушается, т. е. требуется найти такое j ($0 \le j \le i-1$), что $A[j] \le A[i] < A[j+1]$ (при $j=0$ элемент ставится на первое место, а при $j=i-1$ он остается на своем месте). Затем выполняется вставка элемента $A[i]$ на место j. На каждом шаге отсортированная часть массива увеличивается. Для выполнения полной сортировки потребуется выполнить $N-1$ шаг.

Рассмотрим этот процесс на примере.

Пусть требуется отсортировать массив из 10 элементов по возрастанию. Действия на каждом шаге процесса сортировки показаны в табл. 2.3.

Таблица 2.3

1-й шаг	13 6 8 11 3 1 5 9 15 7	Рассматриваем часть массива из одного элемента 13. Вставляем второй элемент массива 6 так, чтобы упорядоченность сохранилась. Так как 6<13, записываем 6 на первое место
2-й шаг	6 13 8 11 3 1 5 9 15 7	Упорядоченная часть массива содержит два элемента (6 и 13). Возьмем третий элемент массива 8 и подберем для него место в упорядоченной части массива. 8>6 и 8<13, следовательно, его место — второе
3-й шаг	6 8 13 11 3 1 5 9 15 7	Следующий элемент — 11. Он записывается в упорядоченную часть массива на третье место, так как 11>8, но 11<13
4-й шаг	6 8 11 13 3 1 5 9 15 7	Далее, действуя аналогичным образом, определяем, что 3 необходимо записать на первое место
5-й шаг	3 6 8 11 13 1 5 9 15 7	По той же причине 1 записываем на первое место
6-й шаг	1 3 6 8 11 13 5 9 15 7	Так как 5>3, но 5<6, то место 5 в упорядоченной части — третье
7-й шаг	1 3 5 6 8 11 13 9 15 7	Место числа 9 — шестое
8-й шаг	1 3 5 6 8 9 11 13 15 7	Определяем место для предпоследнего элемента 15. Оказывается, что этот элемент массива уже находится на своем месте
9-й шаг	1 3 5 6 8 9 11 13 15 7	Осталось подобрать подходящее место для последнего элемента 7
	1 3 5 6 7 8 9 11 13 15	Массив отсортирован полностью

Процедура сортировки:

```
Procedure Sort(N:Integer;Var A:MyArray);
 Var i,j,w:Integer;
 Begin
  For i:=2 To N Do Begin
   w:=A[i];
   j:=i-1;
   While (j>0) And (w<A[j]) Do Begin
     A[j+1]:=A[j];
     j:=j-1;
    End;
   A[j+1]:=w;
 End;
End;
```

Оценим эффективность метода. Для массива, например:
1 2 3 4 5 6 7 8
потребуется $N-1$ сравнение (напомним, что мы сортируем в порядке возрастания), а для массива:
8 7 6 5 4 3 2 1
— $N \cdot (N-1)/2$ или $c=O(N^2)$.

Экспериментальный раздел занятия

Сортировка подсчетом.

Пусть дан массив A из 10 элементов:
10, 5, 11, –5, 1, –4, 13, 12, –4, 13.

Возьмем первый элемент, он больше пяти элементов массива. Запишем 5 в дополнительный массив счетчиков (*Count*). Выполним эту операцию для всех элементов массива A. В итоге получится следующий массив *Count*:
5, 4, 6, 0, 3, 1, 8, 7, 2, 9.

Если разрешается использовать дополнительный массив для хранения отсортированных данных, то остается переписать каждый элемент исходного массива на соответствующее место в результирующем массиве B. То есть первый элемент исходного массива должен оказаться на шестом месте в отсортированном массиве, второй — не пятом и так далее.

```
Procedure Sort(N:Integer;A:MyArray;
                  Var B:MyArray);
  Var i,j:Integer;
      Count:MyArray;
  Begin
   FillChar(Count,SizeOf(Count),0);
   For i:=N DownTo 2 Do
    For j:=i-1 DownTo 1 Do
     If A[i]<A[j] Then Count[j]:=Count[j]+1
      Else Count[i]:=Count[i]+1;
   For i:=1 To N Do B[Count[i]+1]:=A[i];
  End;
```

Естественное желание — убрать массив B. Ниже по тексту приведена соответствующая процедура. Разберите ее. Покажите, что она действительно работоспособна.

```
Procedure Swap(Var x,y:Integer);
  Var w:Integer;
  Begin
```

```
  w:=x;x:=y;y:=w;
 End;
Procedure Sort(N:Integer;Var A:MyArray);
 Var i,j:Integer;
     Count:MyArray;
 Begin
  FillChar(Count,SizeOf(Count),0);
  For i:=N DownTo 2 Do
   For j:=i-1 DownTo 1 Do
    If A[i]<A[j] Then Count[j]:=Count[j]+1
     Else Count[i]:= Count[i]+1;
  For i:=1 To N Do Count[i]:= Count[i]+1;
  For i:=1 To N Do
   While Count[i]<>i Do Begin
    Swap(A[i],A[Count[i]]);
    Swap(Count[i],Count[Count[i]]);
   End;
 End;
```

Предположим, что значения элементов массива *A* принадлежат интервалу [*u*, *v*]. Например, *A* состоит из элементов: 2, 4, 3, 2, 4, 2, 3, 4, 3, 2.

Подсчитаем, сколько раз каждое значение встречается в массиве: двойка — 4 раза, тройка — 3 раза и четверка — 3 раза (массив *Count*: 4, 3, 3). Изменим значения *Count* на 4, 7 (то есть 4+3), 10 (4+3+3). При этих знаниях (*Count*) элементы массива *A* можно сразу записывать на свое место в результирующий массив *B*.

```
Procedure Sort(N:Integer;A:MyArray;
                 Var B:MyArray);
{Считаем, что значения элементов массива A
принадлежат интервалу [1..4].}
 Const u=1;v=4;
 Var i:Integer;
     Count:Array[u..v] Of Integer;
 Begin
  FillChar(Count,SizeOf(Count),0);
  For i:=1 To N Do Count[A[i]]:= Count[A[i]]+1;
  For i:=u+1 To v Do
      Count[i]:=Count[i]+Count[i-1];
  For i:=1 To N Do Begin
```

```
    B[Count[A[i]]]:=A[i];
    Count[A[i]]:=Count[A[i]]-1;
   End;
  End;
```

А можно ли не использовать массив *B*? Оказывается, да. В следующей версии процедуры Sort (ее следует понять и экспериментально проверить) показано, как это сделать.

```
Procedure Sort(t:Integer;Var A:MyArray);
  Const u=1;v=4;
  Var i,j,q:Integer;
      Count:Array[u..v] Of Integer;
  Begin
   FillChar(Count,SizeOf(Count),0);
   For i:=1 To t Do Count[A[i]]:= Count[A[i]]+1;
   q:=1;
   For i:=u To v Do
    For j:=1 To Count[i] Do Begin
     A[q]:=i;
     q:=q+1;
    End;
  End;
```

Задания для самостоятельной работы

1. Измените решения в трех рассмотренных методах так, чтобы осуществлялась сортировка:
   - четных элементов массива;
   - элементов, записанных на нечетных местах;
   - отрицательных элементов массива.
2. Дан массив 12 3 5 7 9 10. За один просмотр методом «пузырька» он становится отсортированным, остальные просмотры ничего не дают. Исключите лишние просмотры.
3. Массив 12 3 5 7 9 10 сортируется методом пузырька за один просмотр, а массив 5 7 9 10 12 3 — за пять (N–1). Напишите программу, реализующую модификацию этого метода, одинаково хорошо (или одинаково плохо?) сортирующую подобные массивы. Устранить «неравноправие» можно путем смены направлений просмотров, то есть первоначально в направлении → получаем 5 7 9 10 3 12, а затем в направле-

нии ← результат — 3 5 7 9 10 12. И так чередуем направления, пока массив не будет отсортирован.

4. Объедините требования заданий 2 и 3 в единое целое. Этот метод называется «шейкер-сортировкой». Реализуйте его.
5. В сортировке простыми вставками уберите переменную x, то есть внутренний цикл запишите в виде:

   While A[0]<A[j] **Do**...

   Подсказка. Массив A необходимо объявить так:

   Array[0..Nmax] **Of Integer**

   и i-й элемент записывать на нулевое место ($A[0]:=A[i]$). Это так называемый прием барьерного элемента.
6. Модификация метода простых вставок. Как вы помните, на момент вставки элемента с номером i элементы массива с номерами от 1 до $i-1$ отсортированы. Выберем из них средний элемент (или один из двух средних) и сравним его с элементом $A[i]$. Если $A[i]$ меньше этого элемента, то поиск места вставки следует продолжать в левой половине, иначе — в правой половине отсортированного массива.

   Эта схема получила название сортировки бинарными вставками. Ее предложил Дж. Мочли в 1946 году в одной из первых публикаций по методам компьютерной сортировки. Реализуйте данный алгоритм.
7. Д. Л. Шелл в 1959 году предложил следующую модификацию метода простых вставок — «сортировку с убывающим шагом». Ее суть покажем на примере.

   Пусть есть массив из 8 элементов. Первоначально делим его на 4 группы по 2 элемента, сортируем элементы в каждой группе. Затем — на 2 группы по 4 элемента, выполняем сортировку для них и, наконец, сортируем одну группу из 8 элементов.

   При простых вставках мы перемещали элемент только в соседнюю позицию, а в этом методе перемещаем на бóльшие расстояния, что приводит к получению более отсортированного массива и, в конечном итоге, к повышению эффективности метода.

   Значения 4, 2, 1 называются приращениями. Последнее приращение должно быть равно 1.

   Д. Кнут дает оценку этого метода при грамотном выборе приращений — O($N^{1.2}$). Лучшим считается выбор приращений в виде взаимно простых чисел. Реализуйте «сортировку с убывающим шагом» для заданных значений приращений.

8. Пусть N является точным квадратом натурального числа, например N=9. Разделим массив на $\sqrt{N}$ групп по $\sqrt{N}$ элементов в каждой. Выберем максимальный элемент в каждой группе... Впрочем, лучше рассмотреть пример.

Дан массив 7 10 3 5 15 9 6 12 8. При первоначальной разбивке получаем следующие группы: (7 10 3) (5 15 9) (6 12 8).

Максимальные элементы 10 15 12.

Максимальный из них 15, он во второй группе. И если оставшиеся элементы второй группы, а это 5 и 9, меньше 10 и 12, то мы нашли сразу три элемента, записываемые на свои места.

Если нет, то заменяем максимальные элементы первой и третьей групп элементами из второй группы.

Этот метод предложен Э. Х. Фрэндом в 1956 году и получил название метода квадратичного выбора. Количество сравнений имеет порядок O($N^{1.5}$). Реализуйте метод.

Занятие № 24. Алгоритмы быстрой сортировки данных

План занятия

1. Сортировка слияниями.

2. Быстрая сортировка.

3. Пирамидальная сортировка.

4. Сортировка 5 элементов за 7 сравнений.

5. Выполнение самостоятельной работы.

Сортировка слияниями

Прежде чем обсуждать метод, рассмотрим следующую задачу. Требуется объединить («слить») упорядоченные фрагменты массива A $A[k]$, ..., $A[m]$ и $A[m+1]$, ..., $A[q]$ в один $A[k]$, ..., $A[q]$, естественно, тоже упорядоченный ($k \le m \le q$).

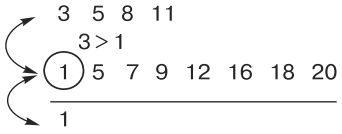
Основная идея решения состоит в сравнении очередных элементов каждого фрагмента, выяснении, какой из элементов меньше, переносе его во вспомогательный массив D (для простоты) и продвижении по тому фрагменту массива, из которого взят элемент. При этом следует не забыть на последнем шаге записать в D оставшуюся часть того фрагмента, который не успел себя «исчерпать» после окончания сравнений.

Пример.

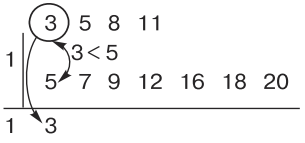
Пусть первый фрагмент состоит из 4 элементов: 3 5 8 11, а второй — из 8: 1 5 7 9 12 16 18 20.

Проиллюстрируем логику работы алгоритма объединения фрагментов.

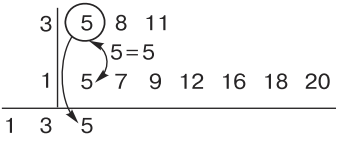
1-й шаг:



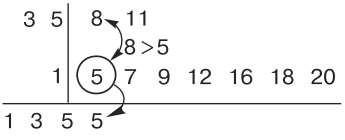
2-й шаг:



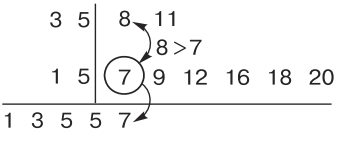
3-й шаг:



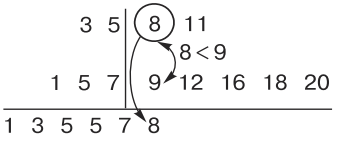
4-й шаг:



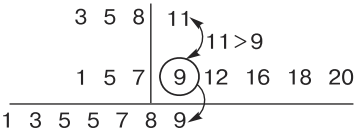
5-й шаг:



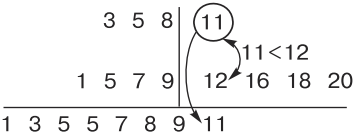
6-й шаг:



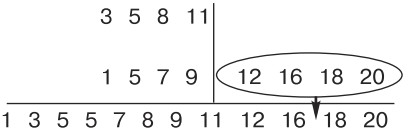
7-й шаг:



8-й шаг:



9-й шаг:



Результат: **1 3 5 5 7 8 9 11 12 16 18 20.**

```
Procedure Sl(k,m,q:Integer;Var A:MyArray);
 Var i,j,t:Integer;
     D:MyArray;
 Begin
  i:=k;
  j:=m+1;
  t:=1;
  While (i<=m) And (j<=q) Do Begin {Пока не
        закончился хотя бы один фрагмент.}
   If A[i]<=A[j] Then Begin
                     D[t]:=A[i];
                     i:=i+1;
```

```
                End
    Else Begin
           D[t]:=A[j];
           j:=j+1;
         End;
    t:=t+1;
  End;
{Один из фрагментов обработан полностью, осталось
перенести в D остаток другого фрагмента.}
  While i<=m Do Begin
               D[t]:=A[i];
               i:=i+1;
               t:=t+1;
              End;
  While j<=q Do Begin
               D[t]:=A[j];
               j:=j+1;
               t:=t+1;
              End;
  For i:=1 To t-1 Do A[k+i-1]:=D[i];
 End;
```

Параметр m из заголовка процедуры можно убрать. Мы «сливаем» фрагменты одного массива A. Достаточно оставить нижнюю и верхнюю границы фрагментов, то есть заголовок процедуры будет выглядеть так:

```
Sl(k,q:Integer;Var A:MyArray);
```

где k — нижняя, а q — верхняя границы.

Вычисление m (эта переменная становится локальной) сводится к присвоению:

```
m:=k+(q-k) Div 2
```

При этом уточнении приведем процедуру сортировки. Первый вызов процедуры — Sort(1,N).

```
Procedure Sort(i,j:Integer);
 Var t:Integer;
 Begin
  If i>=j Then Exit;{Обрабатываемый фрагмент
              массива состоит из одного элемента.}
```

```
    If j-i=1 Then Begin
      If A[j]<A[i] Then Begin{Обрабатываемый
         фрагмент массива состоит из двух элементов.}
                          t:=A[i];
                          A[i]:=A[j];
                          A[j]:=A[i];
                         End;
       End
     Else Begin {Разбиваем заданный фрагмент на два.}
          Sort(i,i+(j-i) Div 2);
          Sort(i+(j-i) Div 2+1,j);
          Sl(i,j,A);
         End;
  End;
```

Замечания для учителя

Метод слияний — один из первых в теории алгоритмов сортировки. Он предложен Дж. фон Нейманом в 1945 году. В логике реализован один из фундаментальных принципов информатики — «разделяй и властвуй».

Рекомендуется проделать с учащимися «ручную» трассировку работы процедуры на различных примерах. Это, во-первых, позволит в очередной раз глубже понять суть принципа и, во-вторых, проверить понимание и усвоение темы «рекурсия».

Эффективность алгоритма, по Д. Кнуту, составляет $c=O(N \cdot \log_2 N)$.

Быстрая сортировка

Описываемый метод предложен Ч. Э. Р. Хоаром в 1962 году. Эффективность метода достаточно высокая, кроме специально подобранных данных, поэтому автор назвал его «быстрой сортировкой».

Изложим идею метода. В исходном массиве A выбирается некоторый элемент x (барьерный элемент). Нашей целью является запись x «на свое место» в массиве, пусть это будет место k, такое, что слева от x были элементы массива, меньшие или равные x, пока еще не упорядоченные, а справа — неупорядоченные элементы, большие x, т. е. массив A будет иметь вид:

$A[1], A[2], \ldots, A[k-1], A[k]=x, A[k+1], \ldots, A[n]$.

В результате элемент $A[k]$ находится на своем месте и исходный массив A разделен на две неупорядоченные части,

барьером между которыми является элемент $A[k]$. Дальнейшие действия очевидны — независимо сортировать полученные части по той же логике до тех пор, пока не останутся части массива, состоящие из одного элемента каждая, т. е. пока не будет отсортирован весь массив.

Пример.

Исходный массив состоит из 8 элементов:
8 12 3 7 19 11 4 16.

В качестве барьерного элемента возьмем средний элемент массива — 7. Произведя необходимые перестановки, получим:
(4 3) 7 (12 19 11 8 16).

Элемент 7 находится на своем месте. Продолжаем сортировку.

Левая часть:

(3) 4 7 (12 19 11 8 16)
3 4 7 (12 19 11 8 16)

Правая часть:

3 4 7 (8) 11 (19 12 16)
3 4 7 8 11 (19 12 16)
3 4 7 8 11 12 (19 16)
3 4 7 8 11 12 (16) 19
3 4 7 8 11 12 16 19

Из этого описания явно просматривается рекурсивная схема реализации, параметрами которой являются нижняя и верхняя границы изменения индексов сортируемой части исходного массива. Приведем процедуру быстрой сортировки из книги классика информатики Н. Вирта.

```
Procedure QuickSort(m,t: Integer);
 Var i,j,x,w: Integer;
 Begin
  i:=m;
  j:=t;
  x:=A[(m+t) Div 2];
  Repeat
   While A[i]<x Do i:=i+1;
   While A[j]>x Do j:=j-1;
   If i<=j Then Begin
               w:=A[i];
               A[i]:=A[j];
               A[j]:=w;
```

```
            i:=i+1;
            j:=j-1;
          End
  Until i>j;
  If m<j Then QuickSort(m,j);
  If i<t Then QuickSort(i,t);
 End;
```

Замечания для учителя

Простая процедура, но возникает «тысяча и один вопрос». Соответствуют ли результаты работы процедуры рассмотренному выше примеру, естественно, при тех же исходных данных? Оказывается, нет. Лучше, если учащиеся убедятся в этом самостоятельно. Результаты работы процедуры приведены в табл. 2.4. Первые два столбца содержат параметры процедуры, третий — массив *A*.

Таблица 2.4

m	*t*	*A*
1	8	4 12 3 7 19 11 8 16
1	8	4 7 3 12 19 11 8 16
1	3	4 3 7 12 19 11 8 16
1	2	3 4 7 12 19 11 8 16
4	8	3 4 7 8 19 11 12 16
4	8	3 4 7 8 11 19 12 16
4	5	3 4 7 8 11 19 12 16
6	8	3 4 7 8 11 12 19 16
7	8	3 4 7 8 11 12 16 19

Продолжим наши рассуждения. В процедуре есть сравнение:

If i<=j **Then** <перестановка элементов A[i] и A[j]>.

Получается, что при равенстве мы выполняем вроде бы ненужную перестановку элементов массива. Но стоит заменить операцию «≤» на «<», как в результате получаем бесконечную работу процедуры («зацикливание»).

Рекурсивный вызов процедуры осуществляется при сравнении:

If m<j **Then** …

А если просто вызвать? Результат не замедлит сказаться. Произойдет переполнение стека адресов возврата.

При изучении темы «Рекурсия» всегда обращается внимание на то, что в таких алгоритмах обязательно должна быть «заглушка». Как она реализована здесь?

Заключительным «аккордом» обсуждения процедуры может быть предложение о написании версии программы, соответствующей примеру, рассмотренному в начале параграфа. Один из вариантов реализации приводится ниже по тексту.

```
Procedure QuickSort(m,t: Integer);
 Var i,j,x,w: Integer;
 Begin
  i:=m;
  j:=t;
  x:=A[(m+t) Div 2];
  While i<=j Do
   If A[i]<x Then i:=i+1
    Else
     If A[j]>x Then j:=j-1
      Else Begin
           w:=A[i];
           A[i]:=A[j];
           A[j]:=w;
           i:=i+1;
           j:=j-1;
          End;
  If m<j Then QuickSort(m,j);
  If i<t Then QuickSort(i,t);
 End;
```

С этой процедурой можно проделать все те же эксперименты, что и с предыдущей программой: изменить операторы сравнения, параметры рекурсивного вызова, условие цикла While. В конечном итоге получается интересное занятие, особенно если совмещать «ручную» трассировку логики с пошаговым выполнением процедур.

Оценим эффективность метода. Предположим, что размер массива равен числу, являющемуся степенью двойки ($N=2^q$), и при каждом разделении элемент x находится точно в середине массива. В этом случае при первом просмотре выполняется N сравнений и массив разделится на две части размерами $\approx N/2$.

Для каждой из этих частей потребуется $N/2$ сравнений и так далее. Следовательно,

$c=N+2 \cdot (N/2)+4 \cdot (N/4)+...+N \cdot (N/N)=O(N \cdot \log_2 N)$.

Впрочем, если N и не является степенью двойки, то оценка будет иметь в среднем тот же порядок.

Пирамидальная сортировка

Метод предложен Дж. У. Дж. Уильямсом и Р. У. Флойдом в 1964 году.

Элементы массива A образуют пирамиду, если для всех значений i выполняются условия: $A[i] \le A[2 \cdot i]$ и $A[i] \le A[2 \cdot i+1]$.

Пример.

Элементы массива 8 10 3 6 13 9 5 12 не образуют пирамиду ($A[1]>A[3]$, $A[2]>A[4]$ и так далее), а элементы массива 3 6 5 10 13 9 8 12 образуют.

Очевидно, что вторая часть элементов массива $A[N/2+1..N]$ независимо от их значений образует пирамиду по той простой причине, что удвоение индекса выводит нас за пределы массива.

Предположим, что нам необходимо отсортировать по невозрастанию элементы массива A:

8 10 3 6 13 9 5 12.

Идею сортировки поясним с помощью табл. 2.5 (курсивом выделена часть элементов массива, образующих пирамиду, жирным шрифтом — отсортированная часть массива). В конечном итоге массив отсортирован.

Таблица 2.5

A	Комментарии
8 10 3 6 *13 9 5 12*	Строим пирамиду из элементов *A* с 1-го по 8-й
8 10 3 *6 13 9 5 12*	
8 10 *3 6 13 9 5 12*	
8 *6 3 10 13 9 5 12*	
3 6 5 10 13 9 8 12	
12 6 5 10 13 9 8 **3**	Меняем местами 1-й и 8-й элементы
12 6 5 *10 13 9 8* **3**	Строим пирамиду из элементов *A* с 1-го по 7-й
12 6 *5 10 13 9 8* **3**	
12 *6 5 10 13 9 8* **3**	
5 6 8 10 13 9 12 **3**	
12 6 8 10 13 9 **5 3**	Меняем местами 1-й и 7-й элементы
12 6 8 *10 13 9* **5 3**	Строим пирамиду из элементов A с 1-го по 6-й

A	Комментарии
...	
9 10 8 12 13 **6 5 3**	Меняем местами 1-й и 6-й элементы
9 10 *8 12 13* **6 5 3**	Строим пирамиду из элементов *A* с 1-го по 5-й
9 *10 8 12 13* **6 5 3**	
8 10 9 12 13 **6 5 3**	
13 10 9 12 **8 6 5 3**	Меняем местами 1-й и 5-й элементы
13 10 *9 12* **8 6 5 3**	Строим пирамиду из элементов *A* с 1-го по 4-й
...	
12 13 **10 9 8 6 5 3**	
13 12 10 9 8 6 5 3	Меняем местами 1-й и 2-й элементы

Пусть мы умеем строить пирамиду из элементов массива *A* от 1 до *q* (процедура Pyram(*q*)), тогда процедура сортировки имеет вид:

```
Procedure Sort;
 Var t,w:Integer;
 Begin
  t:=N;
  Repeat
    Pyram(t);{Строим пирамиду из t элементов.}
    w:=A[1]; {Меняем местами 1-й и t-й элементы.}
    A[1]:=A[t];
    A[t]:=w;
    t:=t-1;
  Until t<2;
 End;
```

Из табл. 2.5 мы видим, что процесс построения пирамиды начинается с конца массива. Последние *q* Div 2 элементов являются пирамидой, поскольку удвоение индекса выводит нас за пределы массива. А затем берется очередной справа элемент и «проталкивается» по массиву до тех пор, пока он не окажется меньше элемента с удвоенным по отношению к нему индексом. Процесс продолжается до тех пор, пока мы не поставим на свое место первый элемент массива. Приведем текст процедуры.

```
Procedure Pyram(q:Integer);
 Var r,i,j,v:Integer;
     pp:Boolean;
 Begin
```

```
    r:=q Div 2+1;{Эта часть массива является
                  пирамидой.}
    While r>1 Do Begin {Цикл по элементам массива,
     для которых необходимо найти место в пирамиде.}
     r:=r-1;
     i:=r; {Индекс рассматриваемого элемента и сам
            элемент.}
     v:=A[i];
     j:=2*i;{Индекс элемента, с которым происходит
            сравнение.}
     pp:=False;{Считаем, что для элемента не найдено
                место в пирамиде.}
     While (j<=q) And Not pp Do Begin
      If j<q Then
       If A[j]>A[j+1] Then j:=j+1;{Сравниваем
              с меньшим элементом.}
      If v<=A[j] Then pp:=True {Элемент находится на
              своем месте.}
       Else Begin {Переставляем элемент и идем
                   дальше по пирамиде.}
            A[i]:=A[j];
            i:=j;
            j:=2*i;
           End;
      End;
      A[i]:=v;
     End;
    End;
```

Метод имеет эффективность порядка $O(N \cdot \log_2 N)$. Однако если рассмотреть вышеприведенную логику, то окажется, что ее оценка $O(N^2 \cdot \log_2 N)$. Почему?

При сортировке процедура Pyram вызывается $N-1$ раз. Эта процедура состоит из двух вложенных циклов сложности N и $\log_2 N$. Итак, упорядочивание одного элемента требует не более $\log_2 N$ действий, построение полной пирамиды $N \cdot \log_2 N$ действий. В вышеприведенной логике после каждой перестановки элементов вновь строится пирамида для оставшихся элементов массива. А зачем? Пирамида почти есть. Требуется только «протолкнуть» верхний элемент на свое место.

Выделим «проталкивание» одного элемента в отдельную процедуру Pr(*r*,*q*), где *r* — номер проталкиваемого элемента, *q* — верхнее значение индекса.

```
Procedure Pr(r,q:Integer);
 Var r,i,j,v:Integer;
    pp:Boolean;
 Begin
  i:=r;
  v:=A[i];{Индекс рассматриваемого элемента и сам
           элемент.}
  j:=2*i; {Индекс элемента, с которым происходит
           сравнение.}
  pp:=False;{Считаем, что для элемента не найдено
             место в пирамиде.}
  While (j<=q) And Not pp Do Begin
   If j<q Then
    If A[j]>A[j+1] Then j:=j+1;{Сравниваем
             с меньшим элементом.}
   If v<=A[j] Then pp:=True {Элемент находится
             на своем месте.}
    Else Begin {Переставляем элемент и идем дальше
              по пирамиде.}
         A[i]:=A[j];
         i:=j;
         j:=2*i;
        End;
  End;
  A[i]:=v;
 End;
```

Какие изменения произойдут с процедурой Sort?

```
Procedure Sort;
 Var t,w,i:Integer;
 Begin
  t:=N Div 2+1;{Эта часть массива является пирамидой.}
  For i:=t-1 DownTo 1 Do Pr(i,N);{Строим
        пирамиду(только один раз).}
  For i:=N DownTo 2 Do Begin
   w:=A[1]; {Меняем местами 1-й и i-й элементы.}
   A[1]:=A[i];
```

```
    A[i]:=w;
    Pr(1,i-1);{Проталкиваем 1-й элемент.}
  End;
End;
```

Сортировка 5 элементов за 7 сравнений

Если использовать один из ранее рассмотренных методов, то количество сравнений при сортировке пятиэлементного массива имеет оценку $5 \cdot \mathrm{Log}_2 5$, а это более десяти сравнений.

Рассмотрим массив A из пяти элементов. Количество возможных комбинаций (различных версий значений элементов массива) равно числу перестановок из пяти элементов, а это $5!=120$. Фактически отсортировать — значит, найти одну комбинацию из 120. Рассуждая с более общих позиций, выясним, какое количество информации I требуется получить для нахождения искомой комбинации (отсортированного массива). В этом случае сравнения элементов массива равносильны вопросам с возможными ответами «да» и «нет». Имеем по известной формуле Р. Хартли $I=\log_2 N!=\log_2 120\approx 7$. Таким образом, пять элементов действительно можно отсортировать за 7 сравнений, а за меньшее количество уже нельзя.

Прежде чем перейти к обсуждению алгоритма, приведем вспомогательную процедуры Swap, с помощью которой по значению индексов i, j переставляются два соответствующих элемента массива.

```
Procedure Swap(i,j:Integer);
  Var t:Integer;
  Begin
    t:=A[i];
    A[i]:=A[j];
    A[j]:=t;
  End;
```

Действия, приведенные на рис. 2.3 (три сравнения), приводят все 120 различных первоначальных ситуаций к виду: $A[1]<A[2]<A[4]$ и $A[3]<A[4]$.

Что мы сделали? Сравнили попарно первые четыре элемента и сравнили максимальные элементы пар. Если максимальный элемент первой пары оказался больше максимального элемента второй пары, то пары переставляются местами. Что осталось сделать? Вставить $A[5]$ в отсортированный массив из

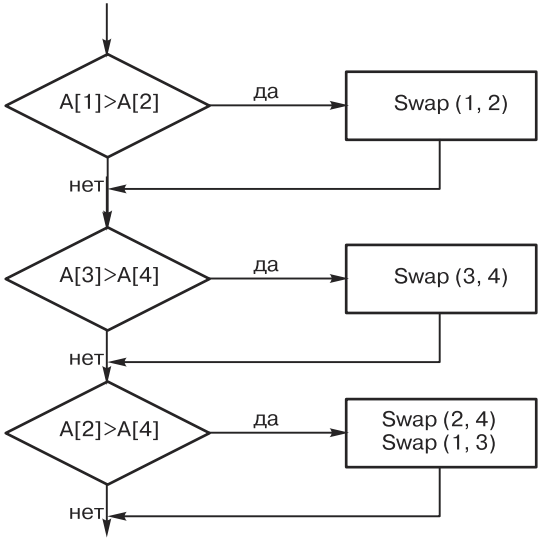


Рис. 2.3. Логика первых трех сравнений при сортировке 5 элементов

трех элементов ($A[1]$, $A[2]$ и $A[4]$). Это можно сделать за два сравнения (рис. 2.4) методом простых вставок. Осталось вставить $A[3]$ в отсортированный массив из четырех элементов

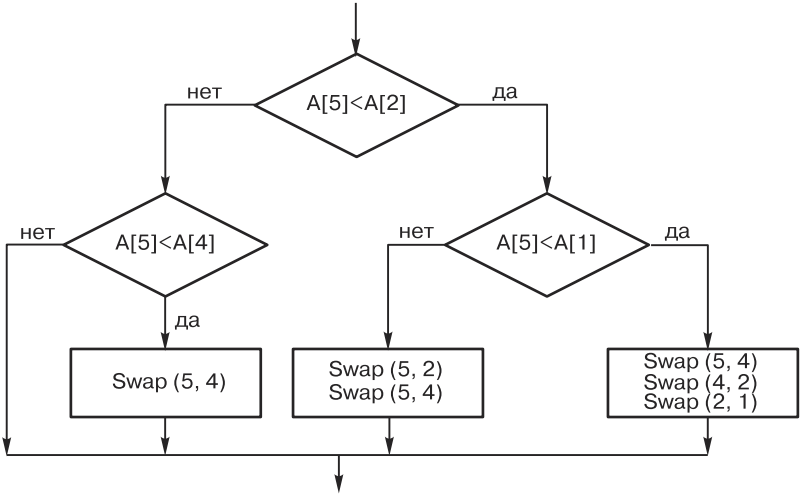


Рис. 2.4. Вставка 5-го элемента в отсортированную часть массива

(*A*[1], *A*[2], *A*[4] и *A*[5]). Так как известно (после первого действия — см. рис. 2.3), что *A*[3]<*A*[4], а второе действие может только изменить его на *A*[3]<*A*[5] (рис. 2.4), то и вставка *A*[3] осуществляется за два сравнения (рис. 2.5).

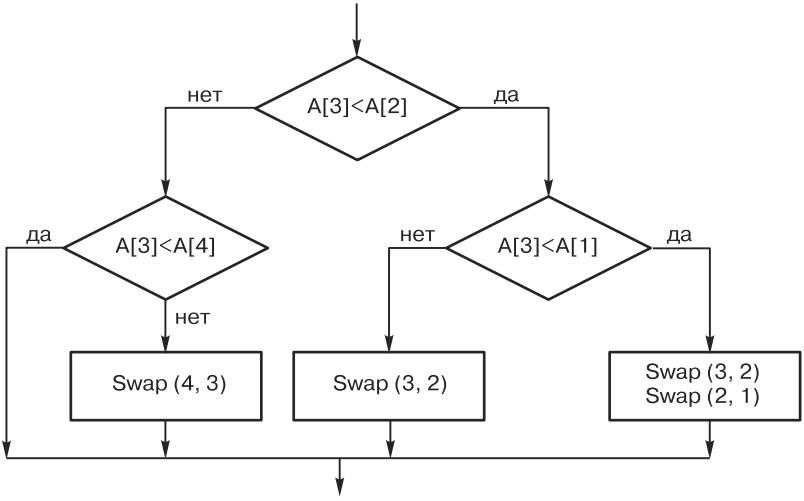


Рис. 2.5. Вставка 3-го элемента в отсортированную часть массива

Рассмотренный алгоритм сортировки пяти элементов предложен Г. Б. Демутом в 1956 году. Ниже приводится фрагмент программной реализации этого алгоритма.

```
Begin
...
  If A[1]>A[2] Then Swap(1,2);
  If A[3]>A[4] Then Swap(3,4);
  If A[2]>A[4] Then Begin
                      Swap(2,4);
                      Swap(1,3);
                    End;
  If A[5]<A[2] Then
    If A[5]<A[1] Then Begin
                        Swap(5,4);
                        Swap(4,2);
                        Swap(2,1);
                      End
```

```
      Else Begin
             Swap(2,5);
             Swap(4,5);
           End
    Else
     If A[5]<A[4] Then Swap(5,4);
   If A[3]<A[2] Then
    If A[3]<A[1] Then Begin
                    Swap(2,3);
                    Swap(1,2);
                   End
     Else Swap(2,3)
    Else
     If A[3]>A[4] Then Swap(3,4);
...
End;
```

Задания для самостоятельной работы

1. Дан массив 7 9 13 1 8 4 10 11 5 3 6 2.

Видим возрастающий участок 7 9 13, а справа, если читать справа налево, есть участок 2 6. Слияние этих двух фрагментов массива дает 2 6 7 9 13. А затем «сливаем» 1 8 и 3 5 11, получаем 1 3 5 8 11. Записываем в массив и получаем: 2 6 7 9 13 4 **10** 11 8 5 3 1.

Обратите внимание на то, как записываем!

Продолжим. Если с первыми фрагментами 2 6 7 9 13 и 1 3 5 8 11 все ясно, то вторые, фрагменты 4 10 и 10, перекрываются. Необходимо учесть сей факт и не дублировать 10 при записи в массив. Получаем: 1 2 3 5 6 7 8 9 11 13 10 4.

Последнее слияние дает 1 2 3 4 5 6 7 8 9 10 11 13. Массив отсортирован. Реализуйте данный алгоритм сортировки.

Этот метод называется у Д. Кнута «сортировкой естественным двухпутевым слиянием». Реализация его с использованием рекурсии доставит много хлопот, а при получении результата — удовлетворение от хорошо сделанной работы.

2. Изменим схему выбора участков для слияния. Будем искать для него не монотонные участки, а участки фиксированной длины («простое двухпутевое слияние»).

Пример.

Исходный массив: 13 1 7 5 4 3 9 10 6 19 14 16 23 2 4 8.
1-й шаг (*длина* 1): 8 13 2 7 4 16 9 19 10 6 14 3 23 5 4 1.
2-й шаг (*длина* 2): 1 4 8 13 3 4 14 16 19 10 9 6 23 7 5 2.
3-й шаг (*длина* 4): 1 2 4 5 7 8 13 23 19 16 14 10 9 6 4 3.
4-й шаг (*длина* 8): 1 2 3 4 4 5 6 7 8 9 10 13 14 16 19 23.

Реализуйте данный алгоритм сортировки. Размер массива не всегда совпадает с числом, равным степени двойки.

3. Интеграция алгоритма простых вставок и простого двухпутевого слияния дает новый вариант алгоритма сортировки.

Разделим массив на участки определенной длины. Каждый из них отсортируем с помощью первого названного алгоритма, а затем используем второй. Так, если брать предыдущий пример, слияние используется на 3-м и 4-м шагах, а первые два заменяются работой по алгоритму простых вставок.

Реализуйте данный алгоритм.

4. Обобщите предыдущие задания заменой слова «двухпутевое» на «трехпутевое» («четырехпутевое», ..., «k-путевое»). Значительное увеличение программного кода при этом недопустимо.
5. Считается, что случайный выбор элемента x в методе быстрой сортировки оставляет его среднюю эффективность неизменной при всех типах массивов (упорядоченный, распределенный по какому-то закону и так далее). Реализуйте эту идею.
6. Замените рекурсивную схему реализации логики метода быстрой сортировки на нерекурсивную.
7. Измените логику работы в методе пирамидальной сортировки так, чтобы элементы массива A сортировались в порядке неубывания.
8. В табл. 2.6 приводятся наименьшие известные значения числа сравнений, достаточных для сортировки последовательностей из N элементов.

Таблица 2.6

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Число сравнений	0	1	3	5	7	10	13	16	19	22	26	30	34	38	42

Разработайте алгоритмы сортировки для нескольких значений N из таблицы, больших 5.

9. Дано N отрезков $[x_i, y_i]$ на прямой (i=1..N). Найдите максимальное k, для которого существует точка прямой, покрытая k отрезками. Временная сложность решения должна быть $O(N \cdot \mathrm{Log}_2 N)$.

Подсказка

Запишите левые и правые концы отрезков в один массив. Сформируйте массив признаков, соответствующий концам отрезков: –1 — левый конец, +1 — правый конец. Отсортируйте первый массив, одновременно переставляя элементы второго массива. Затем при просмотре массива необходимо подсчитывать суммы признаков. Максимальное значение этой суммы есть ответ задачи.

10. Значениями элементов сортируемого массива являются числа 1, 2 и 3. Сортировка элементов по неубыванию выполняется с помощью попарных перестановок элементов. Отсортируйте массив за минимальное количество сравнений.

11. При некоторых исходных данных метод быстрой сортировки работает за время, пропорциональное $O(N^2)$. В этом случае на каждом шаге размер сортируемой части массива должен уменьшаться только на единицу. Попробуем построить такие последовательности для различных значений N. Ограничимся случаем, когда исходными данными является перестановка чисел от 1 до N.

Логику быстрой сортировки можно дополнить счетчиком числа сравнений, генерировать все перестановки как исходные данные и запоминать те, которые дают максимальное число сравнений. А если N — большое число, например 10 000? Для такого «лобового» подхода не хватит мощности любого компьютера. Этот путь решения не подходит. Попробуем по-другому.

При N=3 одной из искомых последовательностей является 1, 3, 2. При первом просмотре 3 переставляется с 2 и рекурсивно вызывается обработка последовательности из двух первых элементов.

При N=4 перестановка элементов на первой итерации должна приводить к тому, что на второй итерации сортируется 1, 3, 2, а это последовательность 1, 4, 2, 3.

Исходя из этого принципа, продолжим «ручное» построение последовательности. Результаты для первых значений N приведены в табл. 2.7.

Таблица 2.7

N	Место в последовательности											
	1	2	3	4	5	6	7	8	9	10	11	12
3	1	3	2									
4	1	4	2	3								
5	1	4	5	3	2							
6	1	4	6	3	2	5						
7	1	4	6	7	2	5	3					
8	1	4	6	8	2	5	3	7				
9	1	4	6	8	9	5	3	7	2			
10	1	4	6	8	10	5	3	7	2	9		
11	1	4	6	8	10	11	3	7	2	9	5	
12	1	4	6	8	10	12	3	7	2	9	5	11

Ваша задача заключается в том, чтобы проверить работоспособность приведенного решения и понять логику его построения.

```
Program sequence;
 Var i,j, N:Integer;
 Begin
  WriteLn(' Длина последовательности ');
  ReadLn(N);
  Write('1',' ');
  i:=2;
  While (2*i<=N) Do Begin
   Write(2*i,' ');
   i:=i+1;
  End;
  If (N>1) And Odd(N) Then Begin
   Write(N,' ');
   i:=i+1;
  End;
  While i<=N Do Begin
   j:=i;
   While Odd(j) Do j:=(j+1) Div 2;
   If j>2 Then Write(j-1,' ')
    Else Write(j,' ');
   i:=i+1;
  End;
  WriteLn;
 End.
```

Занятие № 25. Перебор

План занятия

1. Простые примеры.
2. Расстановка ферзей на шахматной доске.
3. Головоломка судоку.
4. Общая постановка проблемы перебора.
5. Выполнение самостоятельной работы.

Простые примеры

Пример 1.

Автобусные билеты пронумерованы последовательностью из шести цифр, каждая из которых может принимать значение от 0 до 9. Билет называют «счастливым», если сумма первых трех цифр равна сумме последних трех. Необходимо найти количество «счастливых» билетов.

Напрашивается очевидное решение: перебрать все билеты, для каждого из них подсчитать суммы цифр и сравнить их. При совпадении сумм увеличивать значение счетчика количества «счастливых» билетов.

Эту идею можно реализовать различными способами. Фрагмент наиболее простого решения выглядит так:

```
...
k:=0;
For a:=0 To 9 Do
 For b:=0 To 9 Do
  For c:=0 To 9 Do
   For d:=0 To 9 Do
    For e:=0 To 9 Do
     For f:=0 To 9 Do
      If (a+b+c)=(d+e+f) Then k:=k+1;
Write(k);
...
```

Возможна и такая реализация:

```
k:=0;
For a:=0 To 999 Do
 For b:=0 To 999 Do
  If (a Div 100)+((a Div 10) Mod 10)+(a Mod 10)=
     (b Div 100)+((b Div 10) Mod 10)+(b Mod 10)
    Then k:=k+1;
Write(k);
```

Очевидно, что второй вариант реализации является более слабым: восемь операций деления, плюс четыре операции сложения против просто четырех сложений на каждом шаге цикла.

Итак, одним из вариантов формулировки задачи на перебор является требование подсчитать количество объектов, удовлетворяющих определенному условию (или нескольким условиям). Иногда требуется не подсчитать такие объекты, а перечислить или ответить на вопрос об их существовании. В примере объектом является шестизначное число.

Пример 2.

Дано поле 8×8 клеток, на котором находится белая шашка (*W*) и несколько черных (*B*). Определить максимальное число черных шашек, которое белая шашка способна взять за один ход. На рис. 2.6 показана одна из возможных позиций.

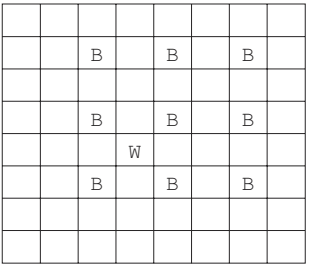


Рис. 2.6. Пример расположения шашек

Идея решения. В общем случае из каждой позиции (без учета граничных) белая шашка может двигаться в четырех направлениях. Для каждого направления следует подсчитать количество черных шашек, которые берутся при этом. А затем из этих четырех значений выбрать максимальное число. Для каждой создавшейся позиции необходимо проверить, возможно ли взятие черной шашки на данном ходе (в данном направлении). Если это невозможно, то анализ в этом направлении закончен. Если взятие возможно, то белая шашка переносится в новую позицию, черная шашка убирается и аналогичные действия начинаются с новой позиции белой шашки. Убирать взятые шашки необходимо, так как если этого не делать, то будет повторно анализироваться одна и та же позиция. При возврате в исходную позицию из следующей (при сделанном ходе) необ-

ходимо возвращать убранную черную шашку на место для того, чтобы перемещение белой шашки в другом направлении осуществлялось с максимально полными исходными данными.

Возможная реализация основной части логики может иметь следующий вид:

```
Const black=2; {Число, которое означает положение
                черной шашки.}
Var pole:Array [1..8, 1..8] Of Integer;
Function Count(i, j:Integer):Integer; {Функция
     подсчета количества черных шашек, которые
     можно взять из клетки с координатами (i,j).}
 Var a:Array [1..4] Of Integer;
    z, k:Integer;
 Begin
  For z:=1 To 4 Do a[z]:=0;
{Четыре одинаковые по своей сути проверки: если в
каком-то направлении есть черная шашка и за ней
свободная клетка, то снять эту шашку и продолжить
решение из новой клетки.}
  If (i>2) And (j>2) And (pole[i-1,j-1]=black)
     And (pole[i-2,j-2]=0) Then Begin
    pole[i-1,j-1]:=0; {Убрали шашку.}
    a[1]:=1+Count(i-2,j-2);
    pole[i-1,j-1]:=black; {Вернули шашку.}
   End;
  If (i<7) And (j<7) And (pole[i+1,j+1]=black)
     And (pole[i+2,j+2]=0) Then Begin
    pole[i+1,j+1]:=0;
    a[2]:=1+Count(i+2,j+2);
    pole[i+1,j+1]:=black;
   End;
  If (i>2) And (j<7) And (pole[i-1,j+1]=black)
     And (pole[i-2,j+2]=0) Then Begin
    pole[i-1,j+1]:=0;
    a[3]:=1+Count(i-2,j+2);
    pole[i-1,j+1]:=black;
   End;
  If (i<7) And (j>2) And (pole[i+1,j-1]=black)
     And (pole[i+2,j-2]=0) Then Begin
```

```
      pole[i+1,j-1]:=0;
      a[4]:=1+Count(i+2,j-2);
      pole[i+1,j-1]:=black;
     End;
    z:=a[1];
    For k:=2 To 4 Do
     If a[k]>z Then z:=a[k];
    count:=k;
End;
```

В данной задаче ведется перебор по всем возможным позициям белой шашки. Проанализировав все варианты в одном направлении, мы возвращаемся в исходную позицию и переходим к другому направлению.

Расстановка ферзей на шахматной доске

На шахматной доске $n \times n$ требуется расставить n ферзей, не атакующих друг друга. Пусть $n=8$ и наша задача заключается в поиске всех возможных расстановок восьми ферзей.

Первый способ решения заключается в последовательном переборе всех возможных расстановок ферзей на 64 клетках. После каждого выбора восьми клеток следует проверить, не нарушается ли условие задачи. Сокращение перебора здесь определяется только тем фактом, что на одну клетку нельзя ставить двух и более ферзей. Число способов расстановки определяется числом сочетаний C_{64}^{8}, а это порядка $4{,}4 \cdot 10^9$ вариантов. Если пренебречь временем проверки условия «атакуют, не атакуют», то и время решения задачи пропорционально этому значению.

Второй способ основан на том, что каждый столбец поля может содержать только одного ферзя. Имеем восемь позиций, в каждой из которых может быть записано число (номер горизонтали) от 1 до 8. Количество таких расстановок равно 8^8 или порядка $1{,}7 \cdot 10^7$.

В третьем варианте решения учитывается тот факт, что и на одну горизонталь можно ставить только одного ферзя. Итак, на первой горизонтали ставим ферзя в любую клетку, их восемь, на второй — в клетки незанятой горизонтали. Таких клеток 7, и тогда на двух горизонталях число расстановок $8 \cdot 7$, а для всей доски — $8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ или $8!=40320$.

В четвертом и последнем варианте учитывается и последнее ограничение: ферзи не могут стоять на одной диагонали.

Количество проверяемых расстановок становится равным 2056.

Для иллюстрации метода перебора рассмотрим доску 4×4 (рис. 2.7). Пусть первый ферзь (рис. 2.7, *а*) поставлен на поле (1, 1) — первая горизонталь (*i*) и первая вертикаль (*j*). Он бьет поля, отмеченные символом «#». Переходим на вторую горизонталь и пытаемся поставить ферзя. Поля (2, 1) и (2, 2) под боем. Ставим ферзя на поле (2, 3). Символом «@» отмечаем свободные поля, которые оказываются и под боем этого ферзя. Переходим на третью горизонталь. Полей не находящихся под боем расставленных ферзей нет.

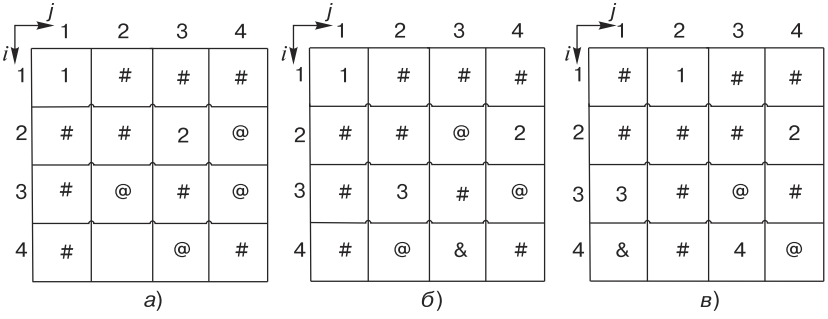


Рис. 2.7. Пример поиска способа расстановки ферзей на доске 4×4

Возвращаемся на вторую горизонталь и пытаемся второго ферзя поставить на следующее свободное поле (см. рис. 2.7, *б*). После этого вновь идем на третью горизонталь. Поле (3, 2) свободно (см. рис. 2.7, *б*). Ставим третьего ферзя и переходим на четвертую горизонталь. Все поля горизонтали под боем. Переходим на третью горизонталь и вновь пытаемся поставить ферзя на следующее свободное поля. Таких нет. Переходим на вторую горизонталь. Результат аналогичный. Переходим на первую горизонталь.

На первой горизонтали выбираем следующие свободное поле (1, 2). Отмечаем символом «#» запрещенные для расстановки поле (см. рис. 2.7, *в*). Переходим на вторую горизонталь и ставим ферзя на единственное свободное поле (2, 4). Переходим на третью горизонталь и ставим ферзя на поле (3, 1). Символом «&» отмечаем свободные поля, которые оказываются

под боем этого ферзя. Последний (четвертый) ферзь ставится на свободное поле (4, 3) — см. рис. 2.7, *в*.

Расстановка ферзей, удовлетворяющая условиям задачи, найдена. Продолжая процесс по этой же схеме перебора, мы найдем все расстановки ферзей на доске.

При реализации рассматриваемой схемы перебора мы идем от одной горизонтали к другой, и если на очередной горизонтали нет свободных полей, то возвращаемся к предыдущей и пытаемся ставить ферзя на следующее свободное поле. Так что два ферзя на одну горизонталь не могут быть поставлены. Для того чтобы не ставить двух и более ферзей на одну вертикаль, следует ввести признак занятости вертикали (массив *Vr*). С диагоналями чуть сложнее. Они двух типов — «восходящие» (рис. 2.8, *а*) и «нисходящие» (рис. 2.8, *б*). Диагонали первого типа характеризуются тем, что для полей, принадлежащих им, сумма индексов i и j постоянна ($i+j$=const). Значение суммы изменяется от 2 до 16. Для полей диагоналей второго типа разность индексов i и j постоянна ($i-j$=const) и изменяется от −7 до 7.

После ввода признаков занятости диагоналей (массивы *Up* и *Down*) постоянно решаемый в ходе перебора вопрос о допустимости постановки ферзя на поле (i, j) становится простым — следует проверить три признака или соответствующие значения массивов *Vr*, *Up* и *Down*.

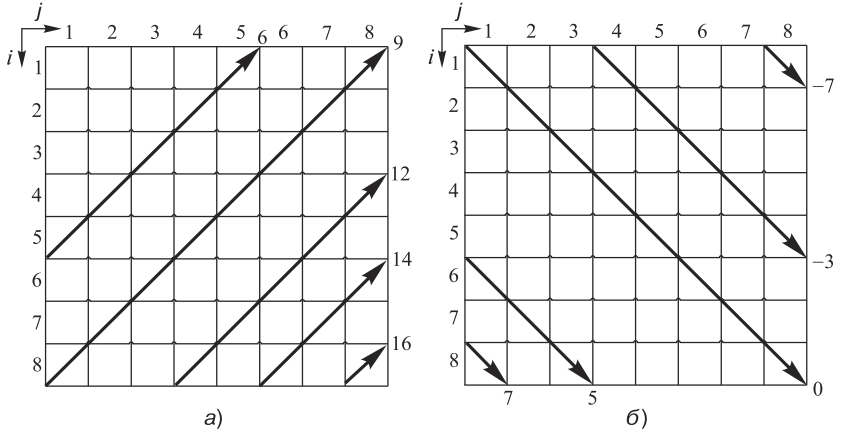


Рис. 2.8. Типы диагоналей

Определим структуры данных:

```
Up: Array[2..16] Of Boolean; {Признак занятости
        диагоналей первого типа.}
Down: Array[-7..7] Of Boolean; {Признак занятости
        диагоналей второго типа.}
Vr: Array[1..8] Of Boolean; {Признак занятости
        вертикали.}
X: Array[1..8] Of Integer; {Номер вертикали, на
   которой стоит ферзь на каждой из горизонталей.}
```

Логика действия — сделать ход или поставить ферзя на поле (*i*, *j*) — сводится к изменению значения признаков (первоначально они все имеют значение True) и фиксации того факта, что на горизонтали *i* ферзь находится на вертикали, определяемой значением индекса *j*.

```
Procedure Hod(i,j:Integer); {Сделать ход.}
 Begin
   X[i]:=j;
   Vr[j]:=False;
   Up[i+j]:=False;
   Down[i-j]:=False;
 End;
```

При возврате от текущей горизонтали к предыдущей ход, сделанный на горизонтали, следует отменить. Это действие опять же сводится к изменению значений признаков.

```
Procedure Ohod(i,j:Integer); {Отменить ход.}
 Begin
  Vr[j]:=True;
  Up[i+j]:=True;
  Down[i-j]:=True;
 End;
```

Проверка допустимости постановки ферзя на поле (*i*, *j*) — это проверка значений признаков.

```
Function Dhod(i,j:Integer):Boolean; {Проверка
        допустимости хода в позицию (i,j).}
 Begin
  Dhod:=Vr[j] And Up[i+j] And Down[i-j];
 End;
```

Реализация перебора и поиска всех способов расстановки ферзей имеет следующий вид:

```
Procedure Solve(i:Integer);
 Var j:Integer;
 Begin
   If i<=8 Then Begin
     For j:=1 To 8 Do
      If Dhod(i,j) Then Begin
        Hod(i,j);
        Solve(i+1);
        Ohod(i,j);
       End;
     End
    Else Begin
      s:=s+1; {Счетчик числа решений, глобальная
               переменная.}
      Print; {Вывод решения (не приводится).}
     End;
 End;
```

Ответом задачи является число 92 — для доски 8×8 существует 92 способа расстановки ферзей.

Головоломка судоку

Классическое судоку. Судоку[14] — это головоломка с числами, ставшая в последнее время очень популярной. В переводе с японского «су» — «цифра», «доку» — «стоящая отдельно».

Игровое поле представляет собой квадрат размером 9×9, разделенный на меньшие квадраты со стороной в 3 клетки. Таким образом, всё игровое поле состоит из 81 клетки. В начале игры клетки частично заполнены, т. е. в них записаны некоторые числа от 1 до 9. В зависимости от того, сколько клеток уже заполнено, конкретную начальную расстановку судоку можно отнести к легким или сложным. На рис. 2.9 приведен пример начального заполнения игрового поля.

[14] Судоку активно публикуют газеты и журналы разных стран мира, сборники судоку издаются большими тиражами. Решение судоку — популярный вид досуга. Иногда судоку называют «магическим квадратом», что в общем-то неверно, т. к. судоку является латинским квадратом 9-го порядка.

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

Рис. 2.9. Пример начального заполнения игрового поля судоку

Правило игры. В судоку есть всего одно правило. Необходимо заполнить свободные клетки цифрами от 1 до 9 так, чтобы в каждой строке, в каждом столбце и в каждом малом квадрате 3×3 каждая цифра встречалась бы только один раз.

Правильно составленная головоломка имеет только одно решение.

Одним из методов поиска решения судоку является перебор всех вариантов заполнения игрового поля.

Алгоритм решения достаточно прост: на каждом шаге расстановки выбирается значение, которое может быть записано в клетку. Если выбранная цифра удовлетворяет правилу, то она записывается в клетку и выбирается цифра для следующей клетки. Если все клетки заполнены, то очевидно, что решение найдено. Если же на очередной клетке ни одно из возможных значений не подходит, следует вернуться к предыдущей клетке и изменить ее значение на следующее допустимое.

Для классического судоку логика анализа возможности записи цифры в клетку заключается в проверках:

- есть ли на данной горизонтали выбранная цифра;
- есть ли на данной вертикали выбранная цифра;
- есть ли в подквадрате выбранная цифра.

Если все проверки показали, что такой цифры нет, то она записывается в клетку.

Один из возможных способов реализации основой логики перебора имеет вид:

```
{Поиск решения для очередной клетки i, j при
заданном n – количестве столбцов и строк поля
головоломки. Нумерация строк и столбцов
начинается с нуля.}
```

```
Function Rec(i, j : Integer): Boolean;
 Begin
  If j = n Then Begin {Проверка: пройдены ли все
       столбцы в строке.}
    j := 0;
    i:=i+1;
   End;
  If i = n Then Rec := True
   Else
    If <Клетка_не_пустая?> Then
      Rec :=Rec(i, j+1) {Переход к следующей
клетке.}
     Else Begin
       m:=<Первое_возможное_значение в клетке.>
       Rec := False;
       While Not Rec And
            m<=<Последнего_возможного_значения> Do
             Begin
         If <Можно_ли_поместить_цифру_m_в_клетку
             (i, j)?>
          Then Begin
           <Поместить_значение_m_в_клетку_(i,j)>;
           Rec :=Rec(i, j+1);
           If Not Rec
            Then <Убрать_значение_m_из_клетки_(i, j)>;
          End;
        If Not Rec
         Then <m:=Следующее_значение_в_клетке>;
       End;
     End;
 End;
```

В данной записи функции Rec использована другая схема разработки. Если в задаче о расстановке ферзей первоначально разрабатывались отдельные (элементарные) процедуры и функции и из них «складывалось» решение, то в данном случае определена общая логика, а фрагменты, подлежащие дальнейшему уточнению, заключены в угловые скобки («<» и «>»).

Общая постановка проблемы перебора

Компьютер может работать только с конечными величинами. Любая задача, решаемая с помощью компьютера, включает множество исходных данных, и оно, в силу сказанного, конечно. Исходные данные в процессе решения по определенному алгоритму анализируются (обрабатываются), и получается нечто под названием результат. Этот процесс может быть описан с помощью следующей модели.

Исходные данные задачи представлены как семейство n конечных упорядоченных множеств A_1, A_2, ... A_n (A_i могут и совпадать). Решение задачи заключается в нахождении вектора $A=(a_1, a_2, ..., a_n)$, где $a_1 \in A_1$, $a_2 \in A_2$, ..., $a_n \in A_n$, удовлетворяющего заданным условиям, или перечислении всех возможных векторов, удовлетворяющих этим условиям.

В первом варианте решения задачи о расстановке ферзей все A_i ($n=8$) состоят из 64 элементов (все клетки шахматной доски), а в четвертом варианте уже из 8 элементов (все клетки одной горизонтали).

В головоломке судоку все A_i для незаполненных клеток состоят из 9 элементов. Значение n равно количеству пустых клеток на момент начала игры.

На стадии предварительной обработки (до фактического перебора вариантов) обычно исследуется структура множеств A_i с целью уменьшения их размерности или установления определенных зависимостей, позволяющих сократить перебор.

Так из A_i (каждое A_i привязано к конкретной клетке) в судоку исключаются все элементы, которые уже есть в строке, столбце и квадрате, соответствующих данной клетке.

Процесс поиска решения можно изобразить графически в виде дерева поиска (рис. 2.10). Корень дерева (нулевой уровень) есть пустой вектор. Его сыновья суть множество кандидатов для выбора a_1 (первый уровень). Следующий уровень — это выбор a_2, при условии что выбрано a_1. И в общем случае вершины k-го уровня являются кандидатами на выбор a_k при условии, что a_1, a_2, ..., a_{k-1} выбраны так, как указывают предки этих вершин. Вопрос о том, имеет ли задача решение, равносилен обходу вершин дерева по приципу от отца к сыну (возможные выборы показаны на рис. 2.10) и поиску совокупности вершин, удовлетворяющих условиям задачи.

В алгоритме перебора вектор A строится покомпонентно слева направо. Предположим, что уже найдены значения пер-

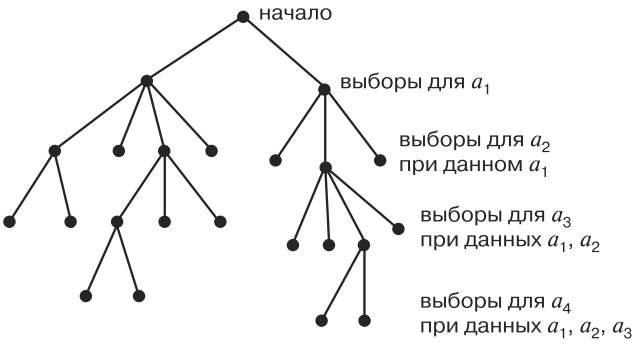


Рис. 2.10. Дерево поиска решения

вых $k-1$ компонент, т. е. $A=\{a_1\, a_2, \ldots, a_{k-1}, ?, \ldots, ?\}$. Тогда заданная совокупность условий ограничивает выбор следующей компоненты a_k некоторым подмножеством S_k множества A_k.

Так в задаче о расстановке ферзей исходные восемь позиций для постановки ферзя на каждой горизонтали уменьшаются до количества позиций, которые находятся не под боем ферзей, поставленных на предыдущие горизонтали.

В судоку запись цифры в клетку приводит к невозможности ее повторения в соответствующих строке, столбце и квадрате. То есть для оставшихся клеток этой строки, этого столбца и этого квадрата из множества цифр — кандидатов на запись исключается записываемая цифра.

Итак, мы вправе выбрать в качестве a_k наименьший элемент S_k и перейти к выбору $k+1$-й компоненты и так далее. Однако если окажется, что S_k пусто, то следует вернуться к выбору $k-1$-й компоненты: отбрасываем a_{k-1} и выбираем в качестве нового a_{k-1} тот элемент S_{k-1}, который непосредственно следует за только что отброшенным. Может оказаться, что для нового a_{k-1} условия задачи допускают непустое S_k, и тогда снова выбирается элемент a_k. Если невозможно выбрать a_{k-1}, то следует вернуться еще на шаг назад и выбрать новый элемент a_{k-2} и т. д.

Формализованный вид этих рассуждений (рекурсивная схема) выглядит следующим образом:

```
Procedure Backtrack (<вектор,i>);
 Begin
  If <вектор является решением> Then <записать его>
```

```
    Else Begin
          <вычислить Si>;
          For <a принадлежащим Si> Do Begin
           <включить a в решение>; {<вектор||a>,
           где символы || означают добавление
           к вектору компоненты.}
           Backtrack (<вектор||a>,i+1) ;
           <исключить a из решения>;
          End;
         End;
End;
```

Оценим временну́ю сложность алгоритма. Пусть количество элементов в каждом из A_i ограничено сверху константой C. Тогда общее количество вариантов, подлежащих перебору, равно C^n. C ростом n эта величина растет по экспоненциальному закону, поэтому в каждой конкретной задаче необходимо искать пути сокращения рассматриваемых вариантов.

Задания для самостоятельной работы

1. В задаче о шашках (простые примеры) в фунции Count четыре раза повторяется практически один и тот же фрагмент кода. Найдите реализацию логики без этого недостатка.
2. Найдите один способ расстановки n ферзей на шахматной доске $n \times n$ клеток так, чтобы они не били друг друга.
3. Путем поворота доски и зеркальных отображений часть расстановок ферзей совпадает. Назовем их симметричными. Найдите способ поиска только несимметричных расстановок.

   Примечание
   Для доски 8×8 таких способов 12.
4. Найдите путь обхода конем доски $n \times m$. Конь должен побывать на каждой клетке только один раз. Начальная позиция коня произвольна.
5. Решите задачу 4, используя следующее правило: новая клетка для коня выбирается не по правилу обхода вариантов просто «по часовой стрелке», а по очередности возрастания количества возможных дальнейших ходов из этой новой клетки. Если таких клеток несколько, то выбирается любая.

6. Магараджа — это фигура, которая объединяет в себе ходы коня и ферзя. Для доски 10×10 найдите способ расстановки 10 мирных (не бьющих друг друга) магараджей.
7. Заполните (пронумеровав позиции) таблицу размером 5×5 числами от 1 до 25. Принцип заполнения: если в клетку с координатами (x, y) записано число i $(1 \le i \le 25)$, то число $i+1$ записывается в клетку с координатами (z, w), вычисляемыми по одному из следующих правил:
   $(z, w)=(x\pm3, y)$;
   $(z, w)=(x, y\pm3)$;
   $(z, w)=(x\pm2, y\pm2)$.
8. Найдите количество всех возможных расстановок чисел в задаче 7 для всех начальных позиций, расположенных в правом верхнем треугольнике таблицы, включая ее главную диагональ.
9. Решите задачу, аналогичную задаче 8, при обходе доски $n\times n$ $(n \le 6)$ шахматным конем. Найдите поле доски, из которого существует минимальное количество обходов.
10. Дано по четыре экземпляра каждого из чисел: 2, 3, 4, 6, 7, 8, 9, 10. Требуется разместить их в таблице 6×6 (рис. 2.11) так, чтобы вместо одинаковых значков стояли одинаковые числа, а суммы чисел на каждой горизонтали, вертикали и обеих диагоналях совпадали.

+		$	#		11
	@			@	
#	@	11	+		#
		$	11	@	
+	!		#	!	$
11		^	+	^	$

Рис. 2.11. Начальная позиция задачи 10

Примечание

Обратите внимание на то, что в таблице уже записано 4 раза число 11.

11. *Задача о магических квадратах.* В клетках квадратной таблицы $n\times n$ $(n \le 5)$ запишите числа 1, 2, 3, ..., $n\cdot n$ так, чтобы их суммы по всем столбцам, строкам и главным диагоналям были одинаковы.

12. *Задача о лабиринте.* Дано клеточное поле $n \times m$, начальная и конечная клетки. Произвольная часть клеток занята препятствиями. За один ход можно перемещаться в одну из свободных клеток по горизонтали или по вертикали. Найдите путь, если он существует, из начальной клетки в конечную.

13. В задаче 12 длина пути измеряется количеством пройденных клеток. Найдите выход из лабиринта минимальной длины.

14. Клетки доски 8×8 раскрашены в белый и черный цвета произвольным образом. За один ход разрешается перемещаться на одну клетку по вертикали или горизонтали. Необходимо пройти из левого нижнего угла в правый верхний, чтобы цвета пройденных клеток перемежались.

15. Решите задачу 14 при условии, что путь должен быть минимальной длины. Под длиной пути понимается количество пройденных клеток.

16. *Задача о рюкзаке.* Даны предметы n различных типов. Количество предметов каждого типа не ограничено. Каждый предмет типа i имеет вес w_i и стоимость v_i, $i=1,2, \ldots n$. Определите максимальную стоимость груза, вес которого равен W.

17. *Задача о коммивояжере.* Классическая формулировка задачи известна уже более 200 лет: имеются n городов, расстояния между которыми заданы; коммивояжеру необходимо, выйдя из какого-то города, посетить остальные точно по одному разу и вернуться в исходный город по маршруту минимальной длины (стоимости).

18. Расставьте на доске $n \times n$ ($n=5$) n ферзей так, чтобы наибольшее число ее полей оказалось вне боя ферзей.

19. Какое наименьшее число ферзей можно расставить на доске $n \times n$ так, чтобы они держали под боем все ее свободные поля?

20. Расставьте на доске как можно больше ферзей так, чтобы при снятии любого из них появлялось ровно одно неатакованное поле.

21. Разработайте полную программную реализацию решения классической судоку.

22. *Судоку X.* В этом варианте судоку вводится дополнительное условие: все цифры должны быть различными также

на обеих основных диагоналях, то есть на каждой из них должны быть записаны все цифры от 1 до 9 (рис. 2.12).

Разработайте решение данного варианта головоломки.

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

Рис. 2.12. Дополнительное ограничение в головоломке судоку X

	15	14	16	14	20	11	18	20	7	
16										15
12										10
17										20
11										16
13										14
21										15
14										10
19										17
12										18
	7	22	16	13	11	21	16	10	19	

	15	14	16	14	20	11	18	20	7	
16	5	2	9	4	7	3	6	8	1	15
12	7	4	1	9	8	6	5	3	2	10
17	3	8	6	1	5	2	7	9	4	20
11	6	1	4	7	3	8	2	5	9	16
13	8	3	2	5	9	4	1	6	7	14
21	9	5	7	6	2	1	8	4	3	15
14	2	9	3	8	6	7	4	1	5	10
19	4	7	8	3	1	5	9	2	6	17
12	1	6	5	2	4	9	3	7	8	18
	7	22	16	13	11	21	16	10	19	

Рис. 2.13. Пример решения головоломки судоку сумм

23. *Судоку сумм*. Здесь в качестве подсказки используются цифры вокруг головоломки, равные сумме трех крайних цифр соответствующих столбца или строки (рис. 2.13).

Разработайте решение данного варианта.

24. *Судоку-астериск*. Здесь вводится дополнительная область из 9 клеток, по одной в каждом из квадратов 3×3. Эти клетки также должны содержать числа от 1 до 9. Пример решения приведен на рис. 2.14, «особые» клетки выделены серым.

Разработайте решение данного варианта.

25. *Гиперсудоку (Windoku, Four-Box Sudoku)*. Этот вариант содержит четыре дополнительных области 3×3 (рис. 2.15), цифры в которых должны удовлетворять общим условиям. Пример решения приведен на рис. 2.15.

Разработайте решение данного варианта.

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

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

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

Рис. 2.14. Пример решения головоломки судоку-астериск

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

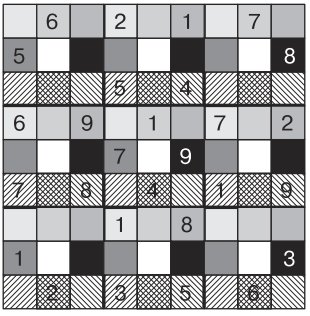
Рис. 2.15. Пример решения головоломки Гиперсудоку

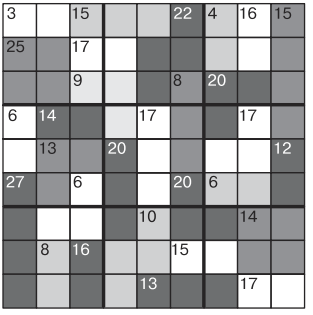
26. *Судоку-отступы*. Введены 9 дополнительных областей по 9 клеток. Каждая такая область состоит из «аналогичных» клеток всех девяти квадратов (т. е. всех левых верхних, всех центральных и т. д.). В каждой группе каждая цифра от 1 до 9 должна встречаться только один раз. Пример решения приведен на рис. 2.16.

Разработайте решение данного варианта.

27. *Судоку-убийца*. В данном варианте задаются дополнительные числа — суммы значений в нескольких произвольных группах клеток общего поля 9×9 (на рис. 2.17 четыре группы таких клеток выделены оттенками серого).

Разработайте решение данного варианта.





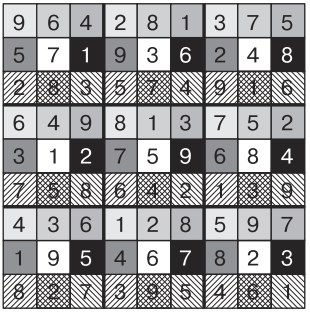


Рис. 2.16. Пример решения головоломки судоку-отступы

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

Рис. 2.17. Пример решения головоломки судоку-убийца

28. *Геометрическая судоку.* Здесь вместо обычных 9 квадратов 3×3 задаются 9 произвольных связных областей из 9 клеток. Пример геометрической судоку и ее решение приведены на рис. 2.18.

Разработайте решение данного варианта.

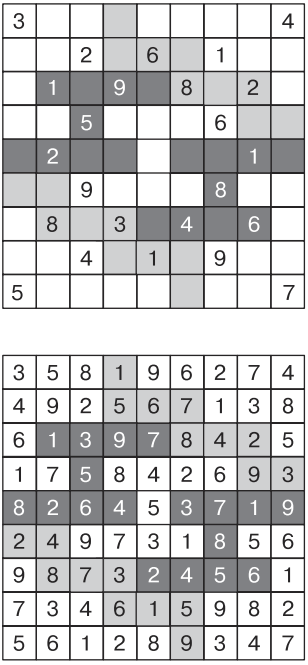


Рис. 2.18. Пример решения головоломки геометрическая судоку

Этюд о программировании

1. О понятии «программа», принципах работы программиста и программировании

Попытаемся ответить на вопрос, что такое программа. Существуют различные, очень умные, научные трактовки этого термина. А есть очень простые. Программа — это «откуда взять, что сделать, куда положить, а если это не так?»[15]. Правда, А. Н. Венц таким образом определяет понятие алгоритма, но, на наш взгляд, оно более соответствует тому, что понимается под словом «программа». Оставим пока в стороне нюансы терминологии. Пусть это будет нашим рабочим определением. Вспомним разобранные программы и ответим на содержащиеся в нем вопросы.

«Откуда взять» — мы пока берем только из файла, в который вводятся данные с клавиатуры компьютера, но есть и другие места, откуда можно взять исходные данные.

«Что сделать» — например, в нашей первой программе имеется один оператор присвоения.

«Куда положить» — ответ очевиден: пока этим местом является то, что связано с монитором нашего компьютера.

И, наконец, четвертый и, наверное, самый главный вопрос — «а если это не так?». Мы убедились, что ответ не очевиден даже в первой программе. Она работает не при всех исходных данных. Если вы откомпилировали, пару раз запустили вашу программу и получили правильные результаты, это еще не значит, что у вас есть работающая программа. Она пока лишь первое приближение к программе. Итак, сформулируем один из основных принципов нашей с вами работы: «Всё под-

[15] А. Н. Венц Профессия — программист. Ростов н/Д: изд-во «Феникс», 1999.

вергай сомнению, всё проверяй сам, ни одного факта на веру». Это относится не только к программам, но и к тому, о чем говорит учитель, к тому, что пишут в книгах по информатике.

Сформулируем еще один из принципов. Работа по схеме воспроизведения, пусть даже творческого, того, что написано в учебнике, или того, что говорит учитель, не приводит к успеху в информатике. Львиная доля успешности освоения предмета приходится на кропотливую самостоятельную работу. А. Н. Венц в своей книге приводит формулу великого программиста (ВП), выведенную экспериментальным путем:

ВП=50% К+30% Т+10% О+5% З+5% ТЛ, где К — знать, КАК это делать, Т — трудолюбие, О — опыт, З — знание, ТЛ — талант.

Итак, на качества вундеркиндов — только 5%, остальное — труд, ежедневный труд. А сейчас... подвергните сомнению эту формулу. Автор, например, не очень понимает разницу между «знать, как» и просто «знанием».

Продолжим уточнение термина «программа». Языков программирования существует великое множество, но на каком бы из них вы ни работали, при создании программы необходимо выполнить следующее:

- ввести данные в программу (*ввод*);
- определить представление этих данных в памяти компьютера (*данные*); точнее, следует говорить о структуре данных;
- определить операции по обработке данных (*операторы*);
- выполнить вывод результатов работы программы (*вывод*).

Организация операций в программе может быть различна:

- некоторые из них выполняются только при определенных условиях (*условия*);
- часть из них выполняется несколько раз (*циклы*);
- часть из них допускает разбивку на блоки и выполняется в различных частях программы (*подпрограммы*).

Итак, эти семь элементов *ввод, данные, операторы, вывод, условия, циклы, подпрограммы* являются основными при создании небольших программ на любом языке программирования.

Однако следует заметить, что рассмотренные уточнения относятся примерно к 1975 году, да и то с некоторыми оговор-

ками. Прежде всего, здесь нет того, что называется **технологией программирования**. Мы не говорим (пока) о том, как из этих элементов «склеивается», собирается программа, а ведь реальные современные программы, по оценкам экспертов, намного сложнее автомобилей. И не только об этом. Абсолютно не затронут вопрос, как программа должна реагировать на события во внешней среде и многое другое. Таким образом, наша трактовка работы программы сводится к некоему последовательному процессу, а это не всегда соответствует действительности. Однако «нельзя объять необъятное», программирование — сложнейший раздел информатики, сложнейшая отрасль производства, а у нас речь идет об «основах программирования».

Но все же обсудим понятие «программирование». Можно дать ему такое определение: «Программирование — теоретическая и практическая деятельность по обеспечению программного управления обработкой данных, включающая создание программ, а также выбор структуры и кодирования данных». Но это очень сложно. Если «деятельность» еще поддается пониманию — это же программирование, а не программа, то программное управление — очень сложно. Попробуем определить иначе, более конструктивно, рассматривая то, чем занимается программист. Есть задача или проблема. В первую очередь программист должен определить возможность ее решения, выбирая соответствующий метод. А затем: разработать проект программы (состоящий из алгоритма на каком-либо из языков программирования); доказать правильность работы программы; предусмотреть возможность внесения изменений в программу.

Таким образом, в укрупненном виде мы видим *три этапа*: до реализации на компьютере, сама реализация и после реализации. Только часть работы связана с выбором структур данных и кодированием — использованием языков программирования. Программирование есть, если так можно выразиться, инженерная работа по конструированию некой целостной системы обработки данных. Отличие программы, например, от любой механической системы в том, что число взаимодействующих частей в программе абсолютно необозримо, и проверить работу этой программной системы, перебрать все возможные способы взаимодействия ее частей немыслимо в разумные сроки даже на сверхбыстродействующих компьютерах. Где же

выход? Только в **технологиях**, обеспечивающих на выходе качественный и надежный продукт.

«Технология — совокупность методов обработки, изготовления, изменения состояния, свойств, форм сырья, материала или полуфабриката в процессе производства или наука о способах воздействия на сырье, материалы или полуфабрикаты соответствующими орудиями производства... ...способ реализации людьми конкретного сложного процесса путем разделения его на систему последовательных взаимосвязанных процедур и операций, которые выполняются более или менее однозначно и имеют целью достижение высокой эффективности. Под процедурой понимается набор действий (операций), посредством которых осуществляется тот или иной главный процесс (или его отдельный этап), выражающий суть конкретной технологии, а операция — это непосредственное практическое решение задачи в рамках данной процедуры, т. е. однородная логически неделимая часть конкретного процесса»[16].

В нашем случае технология должна поддерживать все три этапа работы программиста, о которых речь шла выше, т. к. технология в данном случае есть мастерство изготовления, конструирования систем обработки данных. Обычно, когда говорят о технологиях программирования, имеют в виду лишь второй этап разработки программ, первый и третий не рассматриваются. Действительно, можно считать этот этап решающим, качественный выход на этом этапе обеспечивает достаточную простоту третьего этапа. Третий часто трактуют как этап сопровождения, но мы понимаем его шире — как эволюционное изменение системы обработки данных.

Поговорим о простых и сложных программах. Простыми можно назвать те, что имеют ограниченную область применения, разработаны и сопровождаются одним человеком. Они могут быть профессионально изготовлены, но так или иначе они обозримы, и их сложность не превышает интеллектуальных возможностей человека. Но не все программы таковы. Программы управления железнодорожными или воздушными сообщениями, системы управления базами данных, обеспечивающие параллельный доступ многих пользователей и так далее — это другой класс программ по уровню сложности. Сложность программ — отнюдь не формальный признак, но одно из

[16] Словарь иностранных слов. 19-е изд. – М.: Русский язык, 1990.

важнейших свойств. Но где пролегает граница между простым и сложным? Обычно сложность программы определяется количеством операторов ее исходного текста (табл. 3.1). Это условная градация, простая количественная мера, которую можно принять в качестве первого приближения. Известны разработки программ, содержащих менее 10 000 операторов исходного текста, а по выполняемым функциям относящихся к сложным и даже сверхсложным. Ограничимся интуитивным пониманием простого и сложного. Философский анализ этих понятий применительно к программам — предмет отдельного разговора.

Таблица 3.1

Программа	Количество операторов исходного текста (до)
Простая	1 000
Средней сложности	10 000
Сложная	100 000
Сверхсложная	1 000 000
Гиперсложная	10 000 000 и более

Отметим еще одну особенность больших программ. Они имеют тенденцию к эволюции в процессе их использования. Это часто называют сопровождением, но сопровождение есть устранение ошибок, а эволюция подразумевает внесение в программу изменений, обусловленных изменяющимися требованиями к ней.

Как «бороться» со сложностью? Принцип известен со времен древних римлян — «разделяй и властвуй». Сложную задачу следует разделить на взаимосвязанные подзадачи. Последние, в свою очередь, опять разделяются на свои подзадачи и так далее, вплоть до самых низких уровней нашего понимания задачи. Что мы имеем? Во-первых, сложная задача (проблема, система) описывается некой иерархической структурой, во-вторых, определяются принципы ее декомпозиции и, в-третьих, т. к. каждый уровень — это определенный уровень абстрагирования, создается инструментарий для описания абстракций. Иерархия — это ранжированная, или упорядоченная, система абстракций, расположение частей или элементов целого в порядке от высшего к низшему.

В результате деятельности программиста создается иерархическая структура из абстракций, а «абстракция — это такие

существенные характеристики некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, четко определяют особенности данного объекта с точки зрения дальнейшего рассмотрения и анализа».

Г. Буч[17] разделяет абстракцию сущности объекта (объект представляет собой модель существенных сторон предметной области) и абстракцию поведения (объект состоит из обобщенного множества операций, каждая из которых выполняет определенную функцию). Другими словами, сущность объекта мы описываем на уровне данных в программе, а поведение — действиями над этими данными.

Выскажем и еще один тезис. На наш взгляд, программа — это не что иное, как модель реальной действительности, реальной задачи, причем динамическая модель.

2. Развитие технологий программирования

Сделаем краткий обзор развития технологий программирования, чтобы дать представление о том, какой «виток спирали» развития технологий программирования представлен в книге.

2.1. Операциональное программирование

Этот этап развития технологий программирования характерен для ЭВМ первого поколения (с 1945 по 1959 год). Быстродействие ЭВМ этого поколения — до 50 тысяч арифметических операций в секунду, объем оперативной памяти — в лучшем случае несколько килобайт ячеек. Ресурсы минимальны. Если сравнить эти характеристики с современными компьютерами — миллиарды операций и мегабайты соответственно, то различие потрясает. ЭВМ того времени понимала только цифровые команды, и программа состояла из множества строк, состоящих из цифр, интерпретируемых центральным процессором. Например, 05 825 631 трактовалось как команда сложения двух чисел (код 05), записанных в ячейки с номерами 825 и 631. Минимальные ресурсы ЭВМ требовали строжайшей экономии оперативной памяти и эффективнейших алгоритмов обработки. По

[17] Буч Г. Объектно-ориентированное проектирование с примерами применения. М.: Конкорд, 1992.

взаимосвязи составных частей программа напоминала «спагетти», примерно то, что изображено на рис. 3.1, на котором стрелками показаны команды передачи управления.

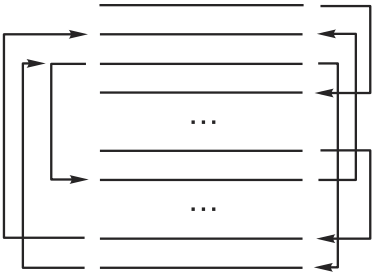


Рис. 3.1. Образ программы: команда (строка) — отрезок прямой линии; передача управления — отрезок прямой линии со стрелкой

Представим программу, состоящую из тысячи таких строк, и отдадим должное программистам того времени. Производительность их работы была очень низкой, так как приходилось вручную распределить все переменные своей программы в оперативной памяти.

Следующий этап развития технологий программирования, связанный с ЭВМ второго поколения, мало отличался от первого. Появились языки программирования типа ассемблера и автокода. Различие, на примере нашей команды сложения, заключалось в том, что она записывалась с использованием мнемоники — ADD (английское «сложить») PR1, ZET, где ADD — код команды, PR1, ZET — имена ячеек. Перевод программы (трансляция), записанной таким образом, в цифровое представление, а только такое понимает ЭВМ, осуществлялся с помощью специальных программ, называемых ассемблерами.

Чем характеризуется этот уровень развития технологий? Его можно назвать о*перациональным программированием*. Первый и третий этапы работы программиста не обсуждаются. Программа «собирается» из мелких деталей, из отдельных операций и имеет достаточно простую структуру, если исключить принцип «спагетти» из управления вычислительным процессом. Уровень абстрагирования — отдельное действие, принципы декомпозиции задачи отсутствуют, во всяком случае о них не говорят. Существует разрыв между требованиями прак-

тики и возможностями программирования. Круг решаемых с помощью ЭВМ задач достаточно ограничен — в основном расчетные работы.

2.2. Нисходящее проектирование, структурное и модульное программирование

Третье поколение ЭВМ (наиболее известная — IBM/360) связано с появлением интегральных схем. Существенной частью ЭВМ становятся операционные системы, на которые возлагаются задачи управления работой компьютера. Операционные системы — ядро системного программного обеспечения. Развиваются языки программирования высокого уровня.

2.2.1. Алгоритмические языки программирования

В 1954 году командой во главе с Джоном Бэкусом (John Backus) из фирмы IBM был создан язык программирования FORTRAN (FORmula TRANslator — транслятор, или переводчик формул), предназначавшийся для описания научных и инженерных задач. Первая версия языка работала еще на ламповом компьютере IBM 704. Этот язык прошел длительный путь развития. Он (точнее, его последние версии FORTRAN 95) используется и в настоящее время. FORTRAN — первый язык программирования высокого уровня. Ключевой идеей, отличающей новый язык от ассемблера, была концепция подпрограмм. Б. Страуструп отмечает: «Использование подпрограмм как механизма абстрагирования имело три существенных последствия. Во-первых, были разработаны языки, поддерживающие разнообразные механизмы передачи параметров. Во-вторых, были заложены основы структурного программирования, что выразилось в языковой поддержке механизмов вложенности подпрограмм, в научном исследовании структур управления и областей видимости. В-третьих, возникли методы структурного проектирования, стимулирующие разработчиков создавать большие системы, используя подпрограммы как строительные блоки»[18].

Если современные компьютеры поддерживают подпрограммы на аппаратном уровне, предоставляя соответствующие команды и структуры данных (стек) прямо на уровне ассембле-

[18] Страуструп Б. Язык программирования C++ 3-е изд./ Пер. с англ. СПб.; М.: «Невский диалект»: «Издательство БИНОМ». 1999.

ра, то в 1954 году этого не было. Перевод программы с языка программирования в машинный код был отнюдь не простой задачей. По признанию Дж. Бэкуса, перед ними стояла задача скорее разработки компилятора, чем языка. Кроме того, синтаксическая структура языка была достаточно сложна для машинной обработки в первую очередь из-за того, что пробелы как синтаксические единицы вообще не использовались. Это порождало массу возможностей для скрытых ошибок, например таких, как приведена далее.

В FORTRAN следующая конструкция описывает «цикл for до метки 13 при изменении индекса от 1 до 50»:

```
DO 13 I=1,50
```

Если же здесь заменить запятую на точку, то получится оператор присвоения:

```
DO13I = 1.50
```

И это не единственный недостаток. Отсутствуют многие привычные языковые конструкции и атрибуты. Компилятор не проверяет синтаксически правильную программу с точки зрения семантической корректности, в нем нет поддержки современных способов структурирования кода и данных и т. д. Понимание самостоятельной ценности языков программирования пришло позже.

В 1959 году в США состоялась специальная конференция, посвященная языкам программирования, в частности языкам программирования для бизнеса. Это собрание получило название CODASYL (от COnference on DAta SYstem Language — конференция по языкам систем обработки данных). Позднее рабочая группа под руководством исполнительного комитета CODASYL выполнила разработку первой версии нового языка программирования COBOL (от COmmon Business-Oriented Language — универсальный язык, предназначенный для бизнеса). На COBOL написаны тысячи прикладных коммерческих систем. Отличительной особенностью языка является возможность эффективной работы с большими массивами данных, что характерно именно в коммерческих приложениях. При этом программа на COBOL напоминает обычный английский текст, это делает ее легко читаемой и упрощает освоение языка. Популярность COBOL была настолько высока, что даже сейчас, при всех его недостатках (по структуре и замыслу COBOL во многом напоминает FORTRAN) появляются новые его диалекты и реализации.

В 1960 году группой разработчиков во главе с П. Науром (Peter Naur) был создан язык программирования ALGOL (ALGOrithmic Language — алгоритмический язык). Этот язык дал начало целому семейству ALGOL-подобных языков (важнейший представитель — Паскаль). В языке нашли отражение новые идеи, относящиеся к блокам и процедурам, он был уникален для своего времени. В 1968 году появилась его новая версия. Она не нашла столь широкого практического применения, как первая, но была весьма популярна в кругах теоретиков.

В 1963 году был создан язык BASIC (Beginner's All-purpose Symbolic Instruction Code — многоцелевой язык символических инструкций для начинающих). Язык задумывался в первую очередь как простой, легко изучаемый язык программирования. BASIC действительно стал языком, на котором учились программировать (плохо это или хорошо — предмет отдельного разговора). Популярность языка подтверждают несколько мощных его реализаций, поддерживающих самые современные концепции программирования (например — Microsoft Visual Basic).

Следует особо сказать о языке PL/1. В 1964 году он был разработан в корпорации IBM для замены COBOL и FORTRAN в большинстве приложений, то есть как универсальный язык. Он обладал огромным количеством синтаксических конструкций. Впервые появилась обработка исключительных ситуаций и поддержка параллелизма. Надо заметить, что синтаксическая структура языка была крайне сложной. Пробелы уже использовались как синтаксические разделители, но ключевые слова не были зарезервированы. В частности, следующая строка — это вполне нормальный оператор на PL/1:

```
IF ELSE=THEN THEN THEN; ELSE ELSE .
```

В силу таких особенностей разработка компилятора для PL/1 была исключительно сложным делом. Язык так и не стал популярен вне мира IBM. В истории программирования была, и, наверное, есть идея создания универсального языка программирования, позволяющего программировать задачи различных классов, другими словами — всевозможные задачи. Обилие языков программирования, а их зарегистрировано более трех тысяч, говорит о тщетности этих попыток.

В 1970 году Н. Вирт создал язык программирования Паскаль. Он получил широкое распространение и поддерживал идеи структурного программирования. Необходимость в опе-

раторе GO TO как инструменте управления порядком выполнения операторов, строго говоря, отпала. Одной из ключевых идей языка является строгая типизация данных. Ее реализация приводит к выявлению большого количества ошибок на стадии компиляции.

В 1972 году появляется язык программирования C. Язык предназначался для разработки операционной системы UNIX, и он позволяет работать с данными практически так же эффективно, как на ассемблере, предоставляя при этом структурированные управляющие конструкции и абстракции высокого уровня (структуры и массивы). Именно с этим связана его огромная популярность и поныне — и именно это является его основным недостатком, ибо компилятор C очень слабо контролирует типы, поэтому очень легко написать внешне совершенно правильную, но логически ошибочную программу.

В 1983 году был создан язык Ада (назван в честь Августы Ады Байрон — графини Лавлейс и дочери английского поэта Байрона, вошедшей в историю вычислительной техники как первый программист). Влияние языка Паскаль на создание данного языка огромно. Главная особенность языка Ада в том, что полностью поддерживается идеология структурного программирования. Есть возможность создавать программы в виде самостоятельных модулей. Язык достаточно сложен, что вызывало и вызывает критику специалистов. Широкого распространения данный язык не получил.

В эти же годы появился ряд языков обработки данных.

1957 — APL (Application Programming Language) для описания математической обработки данных;

1958 — LISP для обработки списков. Многие его особенности унаследованы современными языками функционального программирования;

1962 — Snobol (в 1974 году – его приемник Icon), предназначенный для обработки строк;

1969 — SETL для описания операций над множествами (перечень можно продолжить).

Все вышеназванные языки являются императивными, то есть программы, записанные на них, представляют пошаговое описание решения задачи. Другой подход заключается лишь в постановке проблемы, а поиск решения осуществляет компьютер (декларативная ветвь развития языков программирования, связанная с проблематикой искусственного интеллекта).

В рамках реализации этой идеи существуют два основных подхода: функциональное и логическое программирование. Ключевой идеей первого подхода является запись программы в виде математических формул, а второго — формул математической логики (язык Prolog, 1971 год).

Каждый язык программирования имеет свою историю, о которой можно было бы рассказывать. Остановимся кратко только на одной, ибо подмножество этого языка используется в данной книге. Турбо Паскаль появился на рынке программных продуктов в 1983 году и совершил революцию в программировании. До этих пор предпочтение отдавалось Бейсику — простому, дешевому и легко усваиваемому. Паскаль же был аппаратно зависимым, дорогим и сложным в обращении. С появлением Турбо Паскаля положение меняется. Турбо Паскаль состоит из языка программирования и среды программирования, которая создает удобства в работе.

Изучение Паскаля как языка программирования идет вместе с изучением всей системы Турбо Паскаль. Язык программирования Паскаль был разработан Н. Виртом в 1968-1970 годах и получил широкое распространение благодаря наглядности программ и легкости изучения. Он послужил основой для разработки других языков программирования (например, Ада, Модула-2).

Первая версия Турбо Паскаля использовалась не очень долго — появилась в 1983 году, а уже в 1984 ее заменила вторая версия, которая получила широкое распространение. К осени 1985 года появляется третья версия, более удобная в работе (быстрее работают компилятор и редактор, возможен вызов MS-DOS из программы).

Четвертая версия (1988 год) представила Турбо Паскаль в новом виде (появление новой среды, компилятор стал встроенным). Осенью этого же года разработана пятая версия, у которой еще больше развита среда и появился встроенный отладчик. А в 1989 году появилась версия 5.5, позволившая перейти к объектно-ориентированному программированию.

Шестая версия уже обеспечивала многооконный и многофайловый режим работы, использование «мыши», применение объектно-ориентированного программирования, обладала встроенным ассемблером и имела другие возможности.

В 1992 году фирма Borland International выпустила два пакета программирования на языке Паскаль — это Borland Pascal 7.0 и Turbo Pascal 7.0.

В пакете Turbo Pascal 7.0 использованы новейшие достижения в программировании. Язык этой версии обладает широкими возможностями, имеет большую библиотеку модулей. Среда программирования позволяет создавать тексты программ, компилировать их, находить и исправлять ошибки, компоновать программы из отдельных частей, использовать модули, отлаживать и выполнять отлаженную программу.

2.2.2. Нисходящая технология проектирования программ

Языки программирования Паскаль, С (точнее, языки этих классов) обеспечивают поддержку (начальные этапы) *нисходящей технологии конструирования программ*. Суть нисходящего конструирования программ — в разбивке большой задачи на меньшие подзадачи, которые могут рассматриваться отдельно. Основными правилами для успешного применения данной технологии являются:

1. Формализованное и строгое описание программистом входов функций и выходов всех модулей программы и системы.
2. Согласованная разработка структур данных и алгоритмов;
3. Ограничение на размер модулей.

Нисходящая технология не есть свод жестких правил, скорее, это основной принцип, допускающий вариации в соответствии с конкретными особенностями решаемой задачи.

В свое время в обширной литературе по этому поводу говорилось и о *восходящей* технологии. В этом случае решение (программа) как бы «складывалось из отдельных кирпичиков», из известных решений подзадач. Таким образом, данной технологией оговаривается определенный принцип декомпозиции и иерархическая структура программы. Важнейшей составляющей этой технологии является *структурное программирование*. Первым инициатором структурного программирования был профессор Э. Дейкстра. В 1965 году он высказал предположение о том, что оператор GO TO мог бы быть исключен из языков программирования. Разумеется, структурное программирование представляет собой нечто большее, чем один лишь отказ от оператора GO TO — это некоторые принципы написания программ.

Теоретическими основаниями структурного программирования являются:

1. Формальные системы теории вычислимости (операторные схемы программы А. А. Ляпунова, системы Э. Поста, алгоритмы А. А. Маркова, лямбда-исчисление А. Чёрча).

2. Анализ программ по нисходящей схеме, декомпозиция, основанная на разбивке задач по уровням $0, 1, \ldots, k$.

В классической работе[19] показано, что такая структура (иерархическая, разбитая на уровни) может быть реализована в языке, включающем только базовые управляющие конструкции. Итак, для реализации программ требуются следующие блоки:

- Функциональный блок или конструкция следования (рис. 3.2).
- Конструкция обобщенного цикла (рис. 3.3).
- Конструкция ветвления или принятия двоичного решения, выбора (рис. 3.4).

Рис. 3.2. Конструкция следования

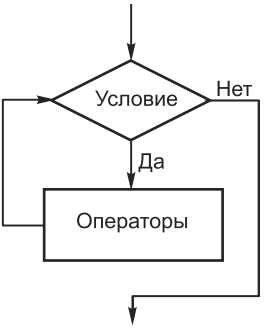


Рис. 3.3. Конструкция цикла

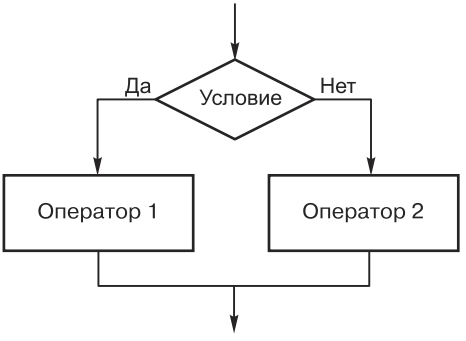


Рис. 3.4. Конструкция ветвления

[19] Böhm C., Jacopini G. Flow diagrams, Turing Machines and Languages with only Two Formation Rules//Communications of the ACM. May 1966.

Характерные черты структурного стиля программирования:

- простота и ясность (программа легко читается и анализируется, достаточное комментирование);
- использование только базовых конструкций;
- отсутствие сетевых структур в программе;
- отсутствие многоцелевых функциональных блоков;
- отсутствие неоправданно сложных арифметических и логических конструкций;
- расположение в строке программы не более одного оператора языка программирования;
- содержательность имен переменных.

Пример неструктурированной логики приведен на рис. 3.5 — сетевая структура. При этом процесс нисходящей разработки программы может продолжаться до тех пор, пока не будет достигнут уровень «атомарных» блоков, то есть базовых конструкций (присвоения, If-Then-Else, Do-While).

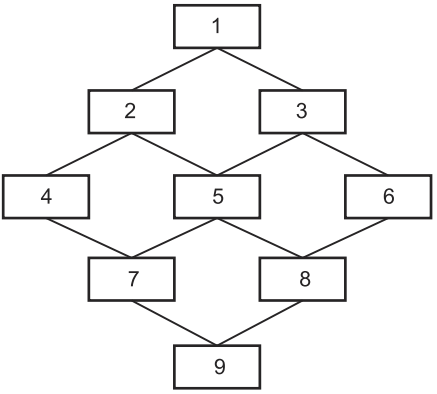


Рис. 3.5. Пример неструктурированной логики (в прямоугольниках цифрами обозначены некие последовательности операторов)

Итак, если формулировать суть в сжатом виде, то в структурном программировании уточнен принцип декомпозиции задачи (в основном ее алгоритмического аспекта, управляющей компоненты, т. е. действий, однако уровень интеграции действий и данных остается «на совести» разработчика) и сделана попытка его строгой формализации.

К нисходящей технологии следует отнести и то, что называется *модульным программированием*. Достаточно независимые фрагменты задачи оформляются как модули. Создаются библиотеки модулей, разрабатывается механизм включения модулей в программу. Модуль должен иметь строго определенный интерфейс и скрытую часть, одну точку входа и одну точку выхода. Если рассматривать модуль как абстрактную концепцию, то ее суть состоит в разбиении пространства имен на две части. Открытая (public) часть является доступной извне модуля, закрытая (private) часть доступна только внутри модуля. Это позволяет рассматривать модуль и как средство борьбы со сложностью программ, и как средство борьбы с дублированием в программировании (то есть как средство накопления и многократного использования программистских знаний). Таким образом, модуль можно рассматривать просто как улучшенный метод создания и управления совокупностями имен и связанных с ними значениями. Схематично структура получаемой программы изображена на рис. 3.6.

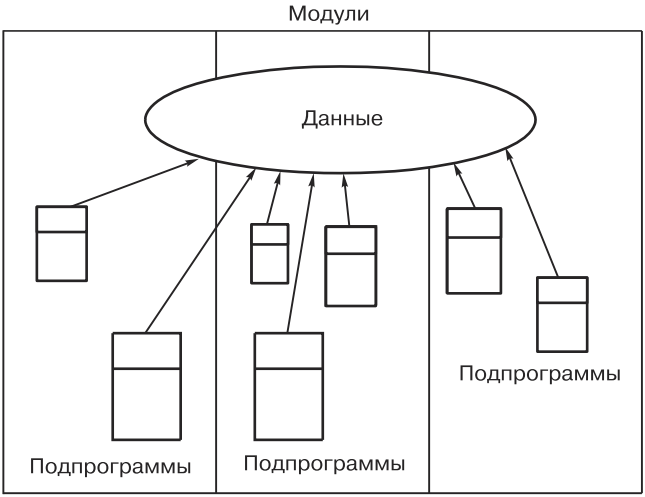


Рис. 3.6. Структура программы при модульном программировании

Из фольклора информатики: «модульность в программировании подобна честности в политике: каждый утверждает, что она — одно из его достоинств, но, кажется, никто не знает, что она собой представляет, как ее привить, обрести или до-

биться». Мы видим очередной этап развития принципов декомпозиции и абстрагирования.

Следует отметить еще одно важное обстоятельство. В информатике существовало и существует как бы два программирования: теоретическое и практическое. Естественно, без теоретического программирования не было бы практического, но если строго следовать первому, то любую часть программы следует строить математическими методами, доказывая правильность ее работы. Вопросы взаимного влияния этих подходов — предмет отдельного исследования. И если теоретическое программирование является уделом математиков-программистов и ему, в принципе, учат (в основном на факультетах вычислительной математики и кибернетики классических университетов), то практическое программирование — это сочетание знаний, интеллекта (аналитических способностей ума) и здравого смысла, другими словами, это искусство разработки программ, причем этому искусству не учат практически нигде. Надо сказать, что на этом витке развития теоретическое программирование оказало очень большое влияние на становление и развитие технологий практического программирования.

2.2.3. Объектно-ориентированное проектирование программ

Очередное поколение компьютеров конструируется на основе БИС и СБИС — больших и сверхбольших интегральных схем. Персональные компьютеры — лакмусовая бумажка прогресса отрасли. Скорости обработки огромны, так же как и объемы оперативной памяти. Избыточность программного кода в несколько тысяч строк не играет принципиальной роли. Технологии программирования, сделав виток, возвращаются на новом уровне к «детской игре в кубики». Но если в период первого-второго поколений программа «собиралась» из отдельных операций и пирамида Хеопса не получалась (поскольку разваливалась), то на этом этапе развития пирамида собирается из объектов — кубиков, интегрирующих в единое целое данные и допустимые действия над этими данными — *объектно-ориентированное программирование*.

Объектно-ориентированные языки программирования

История развития объектно-ориентированных языков программирования в сверхсжатом виде:

1967 — SIMULA (К. Нигард, О. Даль). Первый объектно-ориентированный язык. Произошел от языка ALGOL.

1972 — SMALLTALK (А. Кей). Чистый объектно-ориентированный язык программирования.

1983 — C++ (Б. Страуструп). Расширение языка программирования С. Добавлены проверка типов (прототипы), перегружаемые функции, ссылки, классы, наследование, полиморфизм, спецификаторы шаблона, обработка исключительных ситуаций.

1985 — OBJECT PASCAL (Apple Computer и Н. Вирт). К языку программирования Паскаль добавлены объекты.

1995 — JAVA (Sun Microsystems). Интерпретируемый язык, выполнение программ в виртуальной машине, «сборка мусора», интерфейсы.

Объектно-ориентированные языки программирования характеризуются тремя основополагающими идеями: инкапсуляцией, наследованием, полиморфизмом.

Инкапсуляция. Сочетание данных с допустимыми действиями над этими данными приводит к «рождению» нового элемента в конструировании программы — объекта (элемент абстрагирования). «Рожденный ползать — летать не может» — и наш объект действует только так, как это в нем заложено, и только над тем, что в нем описано. Обращение к данным объекта не через его действия недопустимо.

Наследование. Программист для решения определенного класса задач строит иерархию (систему) объектов, в которой, и это самое главное, каждый следующий производный объект имеет доступ (наследует) к данным и действиям всех своих предшественников («прародителей»). Характер связей между объектами вертикальный.

Полиморфизм. Выделение некоторого действия, т. е. действие должно иметь имя (новый уровень абстрагирования: от управляющих конструкций к абстрактному действию, интегрированному с данными), и создание средств использования действия объектами иерархии. Причем каждый объект реализует это действие так, как оно для него подходит. Пример: есть множество геометрических фигур, образующих иерархию. Действие — перемещение по экрану. Мы видим «скачок» в технологии программирования, впервые действия и данные образуют нечто единое — новый уровень абстрагирования.

Для общей характеристики объектно-ориентированного программирования обратимся к классической работе Г. Буча[20]. Этой технологии присущи определенные принципы абстрагирования и декомпозиции. Задача описывается некой иерархической структурой классов. Основным инструментом построения такой структуры служит реализация концепции наследования. Наследование означает такое соотношение между классами, когда один класс использует структурную или функциональную часть одного или нескольких других классов (соответственно простое и множественное наследование). Иными словами, наследование — это иерархия абстракций, в которой подклассы наследуют строение от одного или нескольких суперклассов.

Логическое завершение в объектно-ориентированных системах получила концепция типизации, которая строится на понятии типов абстрактных данных. «Тип — это точное определение свойств строения или поведения, которое присуще некоторой совокупности объектов». При этом возможно определение как статических, так и динамических связей. Если, например, языку программирования Паскаль присуща строгая типизация, при которой осуществляется контроль на соответствие типам данных и связи статичны во времени, имена связываются с типами во время компиляции и связь не изменяется во время работы программы, то в объектно-ориентированных средах возможна динамическая связь (поздняя связь). Это означает ситуацию, когда тип всех переменных и выражений определяется только во время исполнения программы, что позволяет реализовать идею полиморфизма. Это свойство является самым существенным в объектно-ориентированном программировании наряду со свойством реализации абстракций. Именно это свойство отличает объектно-ориентированное программирование от более традиционных методов программирования с использованием типов абстрактных данных.

Объектно-ориентированные технологии проектирования прикладных программных систем

Выше рассмотрена третья стадия развития объектно-ориентированных технологий решения задач — программирова-

[20] Буч Г. Объектно-ориентированное проектирование с примерами применения. М.: Конкорд, 1992.

ние. Однако она включает и объектно-ориентированный анализ, и объектно-ориентированное проектирование. Коротко остановимся на этих этапах[21].

Проектирование программной системы начинается с анализа требований, которым она должна удовлетворять, в результате чего составляется предварительный проект на основе разработки моделей этой системы. Моделью системы в данном случае называют формальное описание системы, в котором выделены основные объекты, составляющие систему, и отношения между этими объектами. Существуют различные технологии объектно-ориентированной разработки, одна из них — OMT (Object Modeling Techniques). В этой технологии проектируемая программная система представляется в виде трех взаимосвязанных моделей:

1. Объектной модели, которая представляет статические, структурные аспекты системы, в основном связанные с данными.
2. Динамической модели, которая описывает работу отдельных частей систем.
3. Функциональной модели, в которой рассматривается взаимодействие отдельных частей системы (как по данным, так и по управлению) в процессе ее работы.

При этом совокупность моделей системы может быть проинтерпретирована на компьютере (с помощью инструментального программного обеспечения, например Paradigm+).

Построение объектной модели включает:

- определение классов и определение словаря данных;
- определение зависимостей и уточнения атрибутов;
- организацию (с использованием аппарата наследования) системы классов;
- выделение подсистем и определение интерфейсов.

Построение динамической модели включает:

- определение событий, состояний объектов и построение диаграмм состояний;
- формулировку условий, при которых происходит то или иное событие и выполняется переход из состояния в состояние;

[21] Изложение основано на работе С.С Гайсаряна «Объектно-ориентированные технологии проектирования прикладных программных систем» (см. http://www.citforum.ru)

- определение активности состояний объектов и действий, связанных с событиями при переходах объектов из состояния в состояние;
- определение параллельно работающих объектов и синхронизацию (согласование) их работы.

Функциональная модель описывает вычисления в системе и показывает, как выходные данные вычисляются по входным данным. Функциональная модель состоит из набора диаграмм потока данных, которые определяют потоки значений от внешних входов через операции и внутренних схем хранения к внешним выходам. Функциональная модель описывает смысл операций объектной модели и действий динамической модели, а также ограничения на объектную модель.

Объектно-ориентированное проектирование включает в себя разработку архитектуры системы и разработку объектов.

Первая работа включает:

- разбиение системы на модули;
- выявление асинхронного параллелизма;
- распределение модулей и подсистем по процессорам и задачам;
- определение принципов хранения данных;
- управление глобальными ресурсами и программным обеспечением;
- выявление пограничных ситуаций.

При разработке объектов:

- получают операции над классами;
- определяют алгоритмы, реализующие полученные операции;
- оптимизируют пути доступа к данным;
- реализуют управление взаимодействиями с внешними объектами;
- уточняют структуру классов, повышая при этом степень наследования, и определяют зависимости.

Визуальное программирование

В рамках очередного этапа в развитии ЭВМ происходит дальнейшее наращивание их технических характеристик. В начале 1980-х годов в операционных системах появился графический интерфейс (Windows, Linux и др.), который получил широкое распространение к концу десятилетия. Графический интерфейс по своей природе является составной компонентной

системой. Цель его создания состоит не в реализации новых функциональных возможностей, а в том, чтобы наладить связи между графическими элементами управления и функциями внутренних частей приложения. Создан механизм обработки событий (любая программа в процессе своей работы с чем-то или с кем-то взаимодействует).

Визуальную технологию[22] конструирования программ (например, систему программирования Delphi) можно отнести к технологиям этого поколения. Она, во-первых, полностью поддерживает объектно-ориентированную технологию, во-вторых, идеи модульного программирования получают логическое завершение, в-третьих, и это принципиально новое в данной технологии, — создан инструментарий (автоматизация) программирования реакции на события. Структура программного кода вероятностная. Не все маршруты, трассы кодов жестко определены. Элементы программного кода взаимодействуют, начинают работать при возникновении определенных событий. А главное, процесс разработки носит не каскадный, последовательный характер, но развивается по спирали.

Одна из наиболее значительных технологий, появившихся в начале 90-х годов, — Rapid Applications Development (RAD), или быстрая разработка приложений (о ней будет сказано в отдельном параграфе).

Появление визуальных сред быстрой разработки приложений позволило увеличить скорость создания графического интерфейса приложений. Быстрая разработка приложения — это создание макета (прототипа) приложения. Макет приложения строится из компонентов визуальной среды программирования, образующих объектно-ориентированную иерархию. Каждая компонента этой иерархии обладает набором свойств, определенным поведением и своей реакцией на события, происходящие в операционной системе. Создание графического интерфейса приложения аналогично игре «конструктор». Однако создание макета приложения еще не означает, что код программы будет достаточно эффективным и качест-

[22] Термин «визуальная технология» неоднозначен. Он трактуется и как технология разработки приложений в виде графических объектов, с последующим автоматическим переводом их в программный код (широком смысле). В данной работе термин «визуальная технология», «визуальное программирование», используется в узком смысле, как технология с графическим интерфейсом.

венным. Как правило, языки сред быстрой разработки приложений являются интерпретируемыми. Этот факт можно объяснить двумя причинами.

1. В случае применения интерпретации упрощаются компиляторы. В частности, в компиляторе исчезает существенный по сложности компонент генерации машинных кодов. В результате средство быстрой разработки может быть реализовано быстрее. Быстрее можно получить и работоспособный вариант приложения.

2. Применение техники интерпретации машинно-независимых промежуточных кодов снимает (или, по крайней мере, облегчает) проблему переноса на новую платформу как самого средства разработки, так и созданных на его основе информационных систем.

Но в любом случае интерпретатор остается интерпретатором, эффективный программный код получить достаточно сложно. Однако в середине 90-х годов фирмой Borland была разработана среда визуального программирования Delphi, которая обеспечивает создание кода посредством компиляции. Это следующий шаг в развитии средств быстрой разработки.

Выделим отличительные черты средств визуального программирования.

1. Полностью поддерживается объектно-ориентированная технология программирования.

2. Идеи модульного программирования получают логическое завершение.

3. Создан инструментарий программирования реакции на события.

4. Сочетание возможностей быстрого макетирования приложений (программных систем) обеспечивает плавное и предсказуемое развитие проектов любого масштаба.

2.2.4. Дальнейшее развитие технологий

Модель многокомпонентных объектов

Очередной этап развития ЭВМ характеризуется распространением информационных и вычислительных систем и сетей, базирующихся на компьютерах разных классов — от персональных до суперЭВМ.

Для начала 1990-х годов характерно увеличение количества пользователей программ, сложность которых очень сильно возрастает. Необходимость быстрого обновления программ, их

интеграция требуют увеличения количества групп программистов, работающих вместе, что, по теории Ф. Брукса, непременно приводит к уменьшению эффективности работы. Решение этого противоречия было найдено в «концепции третьих фирм». Сторонняя фирма создает часть программы, которую потом могут использовать многие другие фирмы. Однако проблема этого решения состояла в том, что коммерческий интерес не всегда позволял сторонней фирме передавать свой код, обновление версий кода также было проблематичным.

Ответом на эту проблему возникли компонентные технологии: COM (Component Object Model), CORBA (Common Object Request Broker Arhitecture) и др. Компонентные технологии — это промежуточное программное обеспечение объектной среды. Для того чтобы обеспечить взаимодействие объектов и их интеграцию в цельную систему, архитектура промежуточного уровня реализует следующие базовые принципы.

1. Независимость от физического размещения объекта. Компоненты программного обеспечения не обязаны находиться в одном исполняемом файле, выполняться в рамках одного процесса или размещаться на одной аппаратной системе.

2. Независимость от платформы. Компоненты могут выполняться на различных аппаратных и операционных платформах, взаимодействуя друг с другом в рамках единой системы.

3. Независимость от языка программирования. Различия в языках, которые используются при создании компонентов, не препятствуют их взаимодействию друг с другом.

Компонентные технологии — это клиент-серверные технологии, в которых функциональность объекта предоставляется клиенту посредством обращения к абстрактным интерфейсам. Интерфейс определяет набор методов, которые реализуют функции, присущие данному классу объектов. Интерфейс дает клиенту возможность вызывать тот или иной метод, скрывая от него все детали его реализации.

Клиент получает доступ к объекту только путем вызова метода, определенного в интерфейсе объекта. Это означает, что реальные действия выполняются в адресном пространстве объекта, возможно, удаленном по отношению к процессу клиента. Сокрытие деталей реализации позволяет добиться слаженного взаимодействия компонентов вне зависимости от того, где и на какой платформе они реализованы и какой язык программирования для этого использовался.

Взаимодействие между клиентским процессом и процессом, который порождает и обслуживает экземпляры объекта (сервер объекта), использует объектный вариант механизма удаленной процедуры. Механизм удаленной процедуры реализует схему передачи сообщений, в соответствии с которой в распределенном клиент-серверном приложении процедура-клиент передает специальное сообщение с параметрами вызова по сети в удаленную серверную процедуру, а результаты ее выполнения возвращаются в другом сообщении клиентскому процессу.

Для того чтобы реализовать эту схему, на стороне клиента и на стороне сервера поддерживаются специальные компоненты, носящие название клиентский и серверный суррогаты. Чтобы вызвать ту или иную функцию, клиент обращается к клиентскому суррогату, который упаковывает аргументы в сообщение-запрос и передает их на транспортный уровень соединения. Серверный суррогат распаковывает полученное сообщение и в соответствии с переданными аргументами вызывает нужную функцию или нужный метод объекта.

Параметры вызова могут формироваться в отличной от серверной языковой и операционной среде, поэтому на клиентский и серверный суррогаты возлагаются функции преобразования аргументов и результатов в универсальное, не зависящее от конкретной архитектуры представление. Тем самым достигается возможность взаимодействия клиента и сервера на различных платформах.

Следует заметить, что при описании взаимодействия в Computer Sciences разделяют два понятия — «интерфейс» и «протокол». Протокол — это полный набор операций, которые объект выполняет при взаимодействии с другим объектом. Протокол отражает все действия, которым объект может подвергаться сам и которыми может оказать влияние на другие объекты. Интерфейс есть спецификация способа взаимодействия двух сущностей понятным для них обеих образом. Иначе говоря, интерфейс обозначает статику взаимодействия, т. е. что именно, в каком формате и на каком месте должен предоставлять объект, а протокол — динамику взаимодействия — чем именно и в какой последовательности должны обмениваться взаимодействующие объекты.

Соотношение технологии объектно-ориентированного программирования и компонентной технологии. В основе компонентной технологии лежит концепция объектов. Эти бинарные объекты служат строительными блоками, используемыми для

разработки приложений. Объект содержит внутри себя определенную часть функциональности приложения, «выставляя наружу» набор методов и свойств, которые другие приложения и компоненты могут использовать для доступа к этой функциональности.

Идеи объектно-ориентированного программирования нашли свое отражение в компонентной технологии.

Инкапсуляция: данные объекта недоступны его клиентам непосредственно, они инкапсулируются, скрываются от доступа извне. Клиент имеет доступ к данным объекта только через методы интерфейса данного объекта.

Полиморфизм: возможность работы с объектами разных типов, каждый из которых поддерживает данный набор интерфейсов, но реализует их по-разному.

Наследование: имея некоторый объект, можно создать новый, автоматически поддерживающий все или некоторые «способности» старого.

Компонентная технология — это объектно-ориентированная технология, однако способ определения и поведения объектов трактуется иначе.

Повторное использование программных компонент. Основная сложность при использовании объектно-ориентированного программирования — расширение функциональных возможностей. Из-за отсутствия стандартов для компоновки двоичных объектов в единое целое невозможно использовать объекты в различных системах программирования, необходимо распространять объекты вместе с исходным кодом. Компонентные технологии решают эти проблемы.

1. Технология COM есть технология, которая переносит все преимущества объектно-ориентированного программирования, доступные программисту на уровне исходного текста, на двоичный уровень. Реализует концепции объектно-ориентированного подхода и повторного применения кода не на уровне наследования реализации классов внутри одного приложения, а на уровне разных приложений и операционных систем.

2. Компонентные технологии являются общим подходом к созданию всех типов программных сервисов. Они «сглаживают» различия между прикладным и системным программным обеспечением.

3. Безразличен язык программирования: определен двоичный интерфейс, который поддерживают все объекты. Уровень

абстрагирования — интерфейс. Интерфейс определяет уровень доступных методов программной компоненты.

В последнее время продолжением этой линии развития технологий программирования является аспектно-ориентированное программирование[23] (парадигма, предложенная в 90-х годах), которое позволяет разработчику четко разделять задачи, обеспечивать еще большую эффективность инкапсуляции и повторного использования кода. Говорить о нем как о новом скачке в развитии технологий, вероятно, еще преждевременно.

Технологии Active X и OLE

Первая реализация OLE (Object Linking and Embedding — связывание и встраивание объектов) была предназначена для обеспечения механизма создания и работы с составными документами. Элементы, созданные в различных приложениях, например Microsoft Excel и Microsoft Word, интегрируются в рамках единого документа. Составные документы создаются либо связыванием двух разных документов, либо внедрением одного документа в другой.

Частный случай. Дальнейшие версии — разные программные компоненты должны предоставлять друг другу сервисы, по-новому взглянуть на взаимодействие любых типов программ (библиотек, приложений, системного программного обеспечения и др.). Под термином OLE понималось все, что создавалось с использованием парадигмы COM. В 1996 году Microsoft ввела в оборот новый термин ActiveX. Сначала он относился к технологиям, связанным с Интернетом, и к приложениям, выросшим из него, вроде WWW (World Wide Web). Но так как разработки Microsoft были основаны на COM, то ActiveX также была связана с OLE. Все вернулось на круги своя: OLE — означает в настоящее время только технологию создания составных документов связыванием и внедрением, а разнообразные технологии на основе COM, ранее объединенные под названием OLE, называют ActiveX. Технологии ActiveX и OLE — это не что иное, как программное обеспечение, предоставляющее клиентам сервисы через COM-интерфейсы, поддерживаемые COM-объектами. Различные части ActiveX и OLE определяют стандартные интерфейсы для различных целей.

---

[23] Шукла Д., Фелл С., Селлз К. Аспектно-ориентированное программирование// MSDN Magazine. Русская версия. 2002. №1.

Рынок стандартных компонентов. Такую цель и преследует финансируемая Microsoft программа разработки интерфейсов подобного рода — OLE Industry Solutions (Промышленные решения на основе OLE). В рамках этой программы группы финансовых компаний, организаций здравоохранения, поставщиков оборудования для торговых точек и др. определили стандартные интерфейсы компонентов, применяемых в соответствующих областях.

Case-технологии

При проектировании больших программ разработчик начинает с планирования своей работы, рисования некоторых диаграмм, написания каких-то предварительных спецификаций, разработки некоторого макета, позволяющего определить, как все составные части будут взаимодействовать между собой, решать поставленную проблему. В 1960-х годах этот процесс был формализован с помощью блок-схем. Все, кто в то время занимался программированием, проходили через это. Считали, что эти понятия и инструментальные средства можно было использовать не только для разработки программ, но и для описания всех процессов на предприятии. Диаграммные методы играли ключевую роль в разработке программ и в 80-е годы. С их помощью пытались описать все стадии разработки программ. Появились даже специальные рабочие станции для автоматизированного процесса разработки диаграмм. Идея полного описания и контроля разработки получила название программной инженерии (software engineering) — дисциплины, позволяющей строить сложную программу в предсказуемом стиле и с качеством, которое можно измерить и гарантировать. Была сделана попытка автоматизации собственной работы.

Эта старая идея жива и в настоящее время, она имеет только другое название — CASE-технологии (Computer Aided Software Engineering — автоматизация разработки ПО). Первоначально под этим термином понимались средства автоматизации разработки программ. Сейчас он трактуется как программные средства, поддерживающие процессы создания и сопровождения программ, включая анализ и формулировку требований, проектирование, генерацию кода, тестирование, документирование, обеспечение качества и управления проектом разработки. Отношение к CASE-технологиям неоднозначное.

С одной стороны, считают, что CASE-технологии обладают высокими потенциальными возможностями в части увеличения производительности труда, улучшения качества программных продуктов, поддержки унифицированного и согласованного стиля работы. При этом существующие CASE-средства, используя методы структурного и объектно-ориентированного анализа и проектирования программ, имеют различные инструментарии (например, диаграммы, тексты и так далее) для описания внешних требований, связей между компонентами программы, динамики работы. Другая точка зрения заключается в том, что CASE-технологии — не более чем пакеты рисования и конструирования диаграмм.

Истина, вероятно, находится посередине. Д. Васкевич[24] приводит пример из прошлого, связанный с индустрией обработки текстов. В 1980-е годы системы обработки текстов были дорогостоящими, большими и привязанными к конкретным ЭВМ. Эти системы были далеко за пределами возможностей обычных пользователей. Затем появилось следующее поколение для персональных компьютеров, но и они были слишком ограничены и тяжелы в применении. Большинство пользователей не применяли их. В настоящее время программы типа Microsoft Word выполняют больше, чем аналогичные разработки 1980-х годов, и для миллионов людей использование текстовых процессоров стало таким же привычным, как использование ручки.

История повторяется. Программ требуется все больше. Программистам необходимы инструментальные средства проектирования на базе компьютера. Ибо, как писал Г. Буч, «*индустрия программирования по-прежнему важнейшая в мире, и то, что мы делаем, по-прежнему питает мировую экономику. Качество деятельности предприятий, их устойчивость зависят от программных продуктов. Организации, страны – все мы зависим от программного обеспечения, и здесь никаких изменений не предвидится, разве что программные продукты будут играть все большую роль. Разработка программ по-прежнему исключительно сложна. Поэтому мы, профессионалы, должны делать все, чтобы поставлять системы*

[24] Васкевич Д. Стратегия клиент/сервер. Руководство по выживанию для специалиста по реорганизации бизнеса. Киев: Диалектика, 1996.

вовремя и высокого качества: ведь получается, что мир зависит от нас»[25].

Технология быстрой разработки систем RAD

Технология RAD (Rapid Application Development) предназначена для разработки, при активном участии пользователей, информационных систем для бизнес-приложений. RAD призвана обеспечить высокую скорость разработки системы при одновременном повышении качества программного продукта и снижении его стоимости. В технологии поддерживаются четыре этапа работы: анализ и проектирование требований; проектирование; конструирование; внедрение.

Д. Мартин пишет: «В рамках RAD применяются «точные и детальные чертежи и схемы (аналогичные тем, что рисуют конструкторы электронного оборудования) с помощью технологии I-CASE, причем из этих чертежей генерируется программный код. На уровне чертежей выполняется значительная часть проверок. Эти чертежи и схемы весьма эффективны при повседневном общении программистов, системных аналитиков, менеджеров и конечных пользователей. Попытки создавать программы без этих средств означают только одно — безответственное руководство»[26]. Технология I-CASE (Integrated-Computer Aided System Engineering) — это специальный термин, обозначающий интегрированную технологию автоматизированного создания систем, обязательный признак которой — в отличие от обычной, неинтегрированной CASE-технологии — наличие автоматического преобразования чертежей в исходный код нужного языка. В технологии RAD используются разные формы чертежей (схемы «сущность — связь», схемы потоков данных, схемы действий, схемы декомпозиции процессов и так далее). Они необходимы для всестороннего анализа разрабатываемого приложения. Каждый тип схемы — это не что иное, как строго определенный визуальный (графический) язык. Средства I-CASE позволяют устанавливать точные связи между схемами, конвертировать чертежи друг в друга, хранить их в общей базе знаний проекта, создавать компьютерную гиперсхему проекта.

---

[25] Буч Г. О будущем разработки программного обеспечения // MSDN Magazine. Русская версия. 2002. №1.

[26] Martin J. Rapid Application Development. N.-Y.: Macmillan Publishing Co., 1991.

3. Платформа Microsoft .Net Framework, или от Pascal к C#[27]

Начинающие программисты часто задаются вопросом, какой язык программирования им необходимо выучить, чтобы быть востребованными на рынке труда. Ответ на этот вопрос можно сформулировать так: важно не знание языка программирования или технологии, а необходимы понимание фундаментальных принципов программирования и навыки их применения на примере хотя бы одного языка. Однако современные технологии программирования развиваются настолько стремительно, что даже профессионалы не в силах следить за всеми новыми технологиями и детально их изучать. Следовательно, необходимо формировать представление о современных технологиях и платформах программирования.

Традиционно для обучения основам программирования применяется Паскаль в силу простоты синтаксиса и достаточно богатых возможностей. Он позволяет разрабатывать приложения самого разного уровня, однако среди профессионалов он применяется не так часто, при этом следует отметить, что на Западе он практически не используется в крупных компаниях. Раньше наиболее распространенными были либо Java, либо C++. Разработка графического интерфейса на C++ является не самой тривиальной задачей, поэтому часто использовали два языка: C++ для создания «ядра» (часть приложения, включающая в себя основную функциональность и обязанная работать максимально эффективно как с точки зрения быстродействия, так и с точки зрения расхода памяти) и, например, Visual Basic для создания графического интерфейса. Однако сочетание двух языков программирования всегда создавало дополнительные проблемы: приведение типов, передача параметров в функции и так далее. В результате Microsoft начала разработку новой платформы программирования «.Net Framework» (читается как Dot Net), призванную устранить существующие недостатки в процессе разработки бизнес-приложений.

---

[27] Автор благодарит Сергея Юрьевича Иванова за материалы, предоставленные им для написания данного параграфа.

3.1. Общие положения

Ключевая идея платформы .Net Framework заключается в использовании среды исполнения CLR (Common Language Runtime) для компиляции и запуска приложений. В общем работа происходит почти аналогично Java:

- при компиляции проекта в Java создается байт-код, при компиляции проекта под .Net Framework создается MSIL-код (Microsoft Intermediate Language);
- при запуске приложения Java виртуальная машина Java отвечает за его перевод в команды процессора и исполнение, при запуске .Net-приложения CLR осуществляет полное управление приложением: компиляция, выполнение, «сборка мусора» и так далее.

MSIL можно рассматривать как ассембер для платформы .Net Framework, то есть все возможности этой платформы доступны на низкоуровневом языке. Однако создавать приложения на нем неудобно, поэтому при разработке используются языки высокого уровня, такие как: C#, Visual Basic.Net, Managed C++ и другие. C# (читается как C Sharp) был разработан специально под эту новую платформу, при этом в него постарались включить все самое лучшее от языков программирования C++, Java, Pascal, SmallTalk и других. Разработка первой версии языка велась в 1998–2001 гг. под руководством Андерса Хейлсберга[28].

Несмотря на то что был разработан новый язык C#, Microsoft решила не ограничивать платформу только им. Для этой цели была введена спецификация CLS (Common Language Specification), и любой язык, удовлятворяющий CLS, может быть использован для создания .Net-приложений. На сегодняшний день, помимо языков, входящих в стандартную поставку .Net Framework: C#, Visual Basic.Net, Jscript.Net, Managed C++, F#, существует достаточно много языков, для которых написаны компиляторы под платформу .Net Framework: Ada, COBOL, Delphi, Eiffel, FORTRAN, Haskell, IronRuby, IronPython, Lisp, Nemerle, Perl, PHP и другие. Таким образом,

[28] Датский инженер-программист, в 1980 г. написавший свой первый компилятор языка Pascal, который впоследствии был продан компании Borland. Вплоть до 1996 года был главным инженером фирмы Borland и занимался разработкой Delphi. В 1996 г. перешел в Microsoft, где возглавил группу разработчиков языка C#.

программисты могут использовать оптимальный для них язык программирования для создания .Net-приложений.

Кроме того, при разработке одного приложения можно использовать разные языки, т. е. часть проекта может быть написана на C#, а другая часть на Managed C++, при этом не возникнет трудностей во взаимодействии этих частей (как было раньше при использовании обычного C++ и Pascal или Visual Basic). Это достигается за счет еще одной спецификации CTS (Common Type System) — стандартной системы типов, которая обязывает все языки для .Net Framework использовать единые типы данных.

После того как приложение создано и откомпилировано в MSIL-код, его можно запускать, но для этого в системе должна быть CLR (Common Language Runtime) — общеязыковая среда исполнения. При запуске .Net-приложения управление передается CLR, которая полностью контролирует выполнение программы. В частности, при обращении к какому-либо классу или методу CLR инициирует его компиляцию в команды процессора. Самой компиляцией занимается еще один компонент .Net Framework — JIT-компилятор (Just In Time, компиляция «на лету»). К достоинствам такой компиляции можно отнести следующие:

- код компилируется оптимизированным под процессор, установленный на конечном компьютере, а не под процессор, заданный разработчиком;
- код автоматически компилируется как 32- или 64-битный в зависимости от операционной системы.

Недостаток можно отметить такой: время, необходимое на перевод MSIL-кода в команды процессора, тратится во время запуска и работы приложения, что отрицательно сказывается на производительности. Однако здесь следует учитывать, что компиляция происходит постепенно, т. е. не вся программа компилируется при запуске, а отдельные методы по мере обращения к ним (при этом повторная компиляция не требуется, т. к. результаты компиляции сохраняются в памяти).

3.2. История развития

Microsoft .Net Framework — это программная платформа, разработанная компанией Microsoft, для создания приложений и последующего их исполнения.

Разработка данной платформы началась еще в конце 1990-х годов, однако первая версия вышла лишь в 2002 году. На сегодняшний день существуют шесть версий .Net Framework, однако фактически их их всего три. Рассмотрим все версии по порядку.

Версия 1.0 вышла 13 декабря 2002 г., а для разработки приложений была выущена MS Visual Studio .Net. Эта версия не получила широкого распрострения, и сегодня едва ли удастся найти хотя бы одно профессиональное приложение, созданное на ней.

В качестве обновления 24 апреля 2003 г. была выпущена версия .Net Framework 1.1 и новая MS Visual Studio .Net 2003. Это первая версия .Net Framework, которая начала поставляться в составе операционных систем Windows (MS Windows Server 2003) и именно с нее началось широкое использование новой платформы, несмотря на многочисленные споры вокруг нее. К основным недостаткам относили следующие:

- низкая производительность;
- малое количество элементов управления;
- необходимость распространения вместе со своим приложение еще и среды исполнения .Net Framework;
- наличие ошибок в среде исполнения.

7 ноября 2005 г. в свет вышла принципиально новая **версия 2.0**, которая включала в себя намного более широкий набор элементов управления, поддержку 64-битных платформ, новые возможности языков программирования: partial classes (частичные классы, позволяющие хранить код одного класса в нескольких файлах), anonymous methods (анонимные методы — создание метода без объявления его названия, параметров и возвращаемого типа), generics (обобщенные классы — аналог шаблонов в C++). Фактически именно эта версия .Net Framework используется во всех современных .Net-приложениях, т. к. она служит основой для двух следующих версий: 3.0 и 3.5.

Третья версия вышла 6 ноября 2006 г. Ее разработка велась параллельно новой операционной системе Windows Vista. Это версия не была самостоятельной платформой, а использовала полностью .Net Framework 2.0 и добавляла к ней 4 новые библиотеки:

- Windows Presentation Foundation (WPF) — система для создания пользовательского интерфейса, основанная на

XAML (XML-язык для разметки), векторной графике, аппартном ускорении обработки 2D и 3D графики;
- Windows Communication Foundation (WCF) — развитие технолгии Web services — система, основанная на сообщениях, для взаимодействия между приложениями как локально, так и удаленно;
- Windows Workflow Foundation (WF) — технология для определения, выполнения и управления рабочими процессами (workflow);
- InfoCard — технология безопасного хранения персональных данных и унифицированного способа использования их в Интернет.

Наиболее «революционной» из перечисленных является WPF, кардинально меняющая процесс разработки приложений. Она достаточно активно развивается в настоящее время, особенно ее версия Silverlight, предназначенная для Web-приложений.

Версия 3.5 была выпущена 19 ноября 2007 г. Она также является расширением .Net Framework 2.0 и предоставляет новые возможности в языке C# и VB.Net, но центральным звеном в этой версии является LINQ (Language Integrated Query) — язык запросов, напоминающий SQL, который, однако, можно использовать в языках программирования платформы .NET Framework. Вместе с этим появились такие возможности, как:
- анонимные типы;
- методы расширения;
- лямбда-исчисление;
- дерево выражений.

Последняя на сегодняшний день **версия 4.0** стала доступна 10 февраля 2010 г. в качестве Release Candidate. Среди основных изменений можно отметить следующие:
- Parallel Extensions — расширения, предназначенные для упрощения программирования для многопроцессорных и распределенных систем;
- функциональный язык программирования F#;
- средства моделирования Oslo и язык программирования M, предназначенный для создания предметно-ориентированных языков и моделей.

3.3. Сферы применения .Net Framework

Платформа .Net Framework служит своего рода фундаментом для создания технологий самого разного назначения, от создания обычных Windows-приложений и до программирования роботов. Рассмотрим основные типы приложений, которые можно создавать на базе .Net Framework.

3.3.1. Windows-приложения

1. WinForms — это традиционная технология создания Windows-приложения с графическим пользовательским интерфейсом, она мало чем отличается от разработки в среде Delphi, Visual Basic или C++ Builder: в проект добавляется форма, на которую в дизайнере можно «перетаскивать» элементы управления и создавать для них обработчики событий. В версии .Net Framework 1.1 количество элементов управления, поставляемых с Microsoft Visual Studio, было очень мало по сравнению с уже существующими средами, такими как Delphi, для которых были разработаны как стандартные (поставляемые вместе с Delphi), так и элементы управления сторонних компаний. Начиная с Microsoft Visual Studio 2005, этот недостаток был устранен, и на сегодняшний день существует большое количество элементов управления для платформы .Net Framework.

2. Windows services представляет собой средство для создания служб, т. е. приложений, не имеющих графического пользовательского интерфейса и выполняющих свою работу в фоновом режиме. Многие современные приложения строятся с помощью данной технологии: основная функциональность реализуется в виде службы, которая автоматически запускается при запуске Windows, а графический интерфейс для настройки и управления этой службой выносится в отдельный исполняемый файл, который может и вовсе отсутствовать. Подобным образом организованы антивирусные программы, серверы баз данных, например MS SQL Server и др. Разработка служб с использованием .Net Framework мало отличается от создания обычных приложений, единственные отличия — это отсутствие элементов управления и форм, а также необходимость создания специального «инсталлятора» для регистрации службы в операционной системе Windows.

3. WPF, как было уже сказано, — это новая платформа для разработки презентационного уровня приложений, которую

можно считать революционной технологией, поскольку она в корне отличается от WinForms. Назовем основные из этих отличий:

- декларативный подход к разработке GUI;
- принципиально иной подход к созданию и использованию элементов управления;
- расширенные возможности привязки данных (Data binding);
- отделение кода от представления;
- использование векторной графики;
- использование DirectX и аппаратного ускорения для вывода графики.

Несмотря на все достоинства WPF, традиционная WinForms остается и будет оставаться востребованной еще долгое время.

3.3.2. Web-приложения

1. Web Forms, чье название не зря созвучно с WinForms, — разработка Web-приложений, и в самом деле стала очень похожей на создание Windows-приложений: добавляется форма, на нее размещаются элементы управления, точно так же для них создаются обработчики событий и пишется код. Отличия заключаются в специфике Web-приложений: графический интерфейс будет полностью переведен в стандартный HTML-код при запросе страницы пользователем, обработка событий осуществляется на сервере.

2. AJAX (Asynchronous Javascript and XML) — это не технология Microsoft, а общий подход «фонового» обмена данными браузера с Web-сервером, что позволяет пользователю без полной перезагрузки страницы получать ответную реакцию сайта на свои действия. Microsoft предоставила свою библиотеку элементов управления AJAX, которую можно применять при создании Web-приложений.

3. Silverlight — новая платформа для разработки RIA-приложений (Rich Internet Application), предоставляющая богатый пользовательский интерфейс и гораздо большие возможности по сравнению с традиционными HTML-страницами. Данная технология сначала появилась в составе WPF и называлась XBAP (XAML Browser APplication), сегодня уже существует бета-версия Silverlight.

4. К сожалению, Silverlight-приложения поддерживаются далеко не всеми современными браузерами: полная поддержка

реализована лишь в Internet Explorer и Firefox, Safari и Google Chrome поддерживают только версии 1.0 и 2.0, а Opera имеет лишь неофициальную поддержку данной технологии.

3.3.3. Коммуникационные приложения

1. .Net Remoting — это программный интерфейс от Microsoft для межпроцессного взаимодействия. Для клиента вызов того или иного метода удаленного объекта ничем не отличается от работы с локальными объектом, при этом вся работа (сериализация данных, передача их на сервер и обратно) осуществляется специальным объектом Channel, который работает поверх TCP, HTTP или именованных каналов. Данная технология пришла на смену технологиям COM и DCOM, однако сегодня в качестве замены .Net Remoting предлагается новая WCF, появившаяся вместе с выходом .Net Framework 3.0.

2. ASP.NET Web Services — технология Web-служб от Microsoft. Здесь можно провести аналогию: ASP.NET Web Services так же соотносится с Web Forms, как Windows Services и WinForms, то есть это неграфическое приложение, имеющее API для взаимодействия.

3. WCF представляет собой новую упрощенную унифицированную программную модель межплатформенного взаимодействия. В ее основу легли .Net Remoting и ASP.NET Web Services.

3.3.4. Приложения для мобильных устройств

Существует «урезанная» версия .Net Framework — .Net Compact Framework, которая разработана для запуска приложений на устройствах, основанных на платформе Windows CE, таких как PDA, мобильные телефоны, заводские контроллеры, и др. Сегодня существует поддержка следующих систем:

- Windows Mobile;
- Windows Embedded;
- Symbian.

3.3.5. Игровые приложения

Существует специальный набор инструментов для создания игровых приложений — Microsoft XNA, включающий два основных компонента.

1. XNA Framework представляет собой модификацию .Net Framework, оптимизированную для игр. Таким образом игра

может запускаться в любой среде, поддерживающей XNA Framework, например работать как на игровой консоли Xbox, так и на обычном компьютере под управлением Windows;

2. XNA Game Studio — это среда разработки для платформы XNA Framework. На сегодняшний день используется версия XNA Game Studio 3.1, анонсированная в марте 2009 г., а недавно стала доступна XNA Game Studio 4 CTP.

3.3.6. Приложения для программирования роботов

Robotics Developer Studio 2008 — среда для управления роботами и их симуляции. Robotics Developer Studio основана на библиотеке CCR (Concurrency and Coordination Runtime), .NET-реализации библиотеки для работы с параллельными и асинхронными потоками данных, используя обмен сообщениями, и DSS (Decentralized Software Services) — облегченном средстве создания распределенных приложений на основе сервисов, которое предусматривает управление множеством сервисов для корректировки поведения в целом.

Выводы

1. На этапе развития **операционального программирования** программа «собирается» из мелких деталей, отдельных операций и имеет достаточно простую структуру, если исключить принцип «спагетти» из управления вычислительным процессом. Уровень абстрагирования (*формального описания*) — отдельное действие, принципы декомпозиции задачи отсутствуют, во всяком случае, о них не говорят. Системный синтез осуществляется, скорее, на интуитивном уровне. Хаос, творящийся в программе, приводит к непониманию того, как она вообще работает. Программа сверхоткрыта, от внешних воздействий не защищены любые ее части.

2. На этапе развития **структурного программирования** и **нисходящей технологии проектирования программ** выработан ряд ключевых идей. В распоряжение разработчиков предоставлены строгие формализованные методы описания программ и принимаемых технических решений. Используется наглядная графическая техника (схемы, диаграммы). Однако труд этот не был автоматизирован, а вручную невозможно разработать и графически представить строгие формальные спецификации программы, проверить их полноту и непротиворе-

чивость и тем более изменить. Идеи Э. Дейкстры реализованы в полной мере, что позволило сделать скачок в развитии технологий: от отдельной инструкции к подпрограмме, модулю, которые обладают определенной автономией. Рассматриваются механизмы взаимодействия составных частей программы, они строго оговариваются. Появилась возможность реализации рекурсии. Концепция типа данных (*информационное описание задачи*) послужила базисом для конструирования сложных данных (*сложных информационных структур*), и возникают предпосылки для решения задач в терминах самой задачи. Уровень абстрагирования (*формализации*) повысился. Нисходящие и восходящие технологии являются основой как системного анализа, так и синтеза программных решений. Для «борьбы со сложностью» появляется инструментарий. Осознана необходимость устранения сверхоткрытости, появляются первые методы ее устранения.

3. **Объектно-ориентированное программирование**, точнее, объектно-ориентированное проектирование — это новый уровень проектирования информационных систем. Ключевая идея — интеграция в единое целое данных и действий над данными (*информации и управления*). Получили дальнейшее развитие методы абстрагирования (*формализации*). Для «борьбы со сложностью» разработаны новые методы — строгая дисциплина взаимодействия объектов. Системный синтез (сборка в единое) получает определенные правила действий. Открытость программы регулируется, в частности, за счет скрытости части данных объекта и методов работы с ними, появляется определенная логика задания степени открытости. Превалирует каскадная модель конструирования программы, степень нелинейности в разработке возрастает, и нелинейность, если так можно выразиться, функционирует по определенной логике.

4. На этапе развития **визуального программирования** создается инструментарий для поддержки нелинейного характера процесса разработки программ. Идея получила логическое завершение. Работа начинается с изготовления действующего макета программы (инструментарий системного синтеза). Степень открытости регулируется разработчиком. Для «борьбы со сложностью» появились новые механизмы, они следуют из процесса сборки программ из компонент. Методы абстрагирования (*формализации*) охватывают такую сложную часть управления вычислительным процессом, как обработка реак-

ций на события, что позволило упростить, в частности, программирование интерфейсных задач. На этом этапе, может быть впервые, достигнуто единство методов: поддержки нелинейности процесса реализации проектов, определения открытости программ и механизмов противодействия сложности.

5. На следующем этапе идет дальнейшее интенсивное развитие методов системного синтеза и анализа. Анализу и, следовательно, автоматизации подвергаются все виды деятельности программиста.

6. Интеграционные процессы по разработке различных приложений (программных продуктов) усиливаются. Создаются единые платформы для этих целей, примером которой может быть .Net Framework.

Минимальные системные требования определяются соответствующими требованиями программ Adobe Reader версии не ниже 11-й либо Adobe Digital Editions версии не ниже 4.5 для платформ Windows, Mac OS, Android и iOS; экран 10"

Учебное электронное издание

Серия: «Развитие интеллекта школьников»

Окулов Станислав Михайлович

ОСНОВЫ ПРОГРАММИРОВАНИЯ

Редактор *Е. В. Баклашова*
Художник *Н. А. Новак*
Технический редактор *Е. В. Денюкова*
Корректор *Е. Н. Клитина*
Компьютерная верстка: *Е. А. Голубова*

Подписано к использованию 25.09.19.
Формат 125×200 мм

Издательство «Лаборатория знаний»
125167, Москва, проезд Аэропорта, д. 3
Телефон: (499) 157-5272
e-mail: info@pilotLZ.ru, http://www.pilotLZ.ru

РАЗВИТИЕ ИНТЕЛЛЕКТА ШКОЛЬНИКОВ

Станислав Михайлович ОКУЛОВ

Декан факультета информатики Вятского государственного гуманитарного университета, кандидат технических наук, доктор педагогических наук, профессор, почетный работник высшего профессионального образования РФ. Автор 9 изобретений по элементам ассоциативных вычислительных структур и автор (соавтор) 15 книг по информатике для школьников и студентов.

Книги автора, вышедшие в серии «Развитие интеллекта школьников»:

«Программирование в алгоритмах»;
«Ханойские башни»;
«Абстрактные типы данных»;
«Алгоритмы обработки строк»;
«Динамическое программирование».

Область интересов: развитие интеллектуальных способностей школьника при активном изучении информатики, методика преподавания информатики в школе и вузе.

С 1993 по 2003 год деятельность в вузе совмещал с работой учителя информатики. За это время его ученики отмечены 33 дипломами (1-й и 2-й степени) на российских олимпиадах школьников по информатике; трое из них представляли Россию на международных олимпиадах.

ОСНОВЫ ПРОГРАММИРОВАНИЯ

В книге рассмотрены фундаментальные положения программирования: конечная величина и конструируемые на ее основе различные типы данных; управляющие конструкции – элементарные составляющие любого алгоритма и основа управления вычислительным процессом; структуризация задач как основополагающий механизм их реализации на компьютере; упорядочение (сортировка) как основа эффективной работы с любыми данными и, наконец, перебор вариантов как универсальная схема компьютерного решения задач.

Для учащихся старших классов,
студентов и учителей информатики.