А. С. Мирончено

Императивное и объектно-ориентированное программирование на Turbo Pascal и Delphi

Глубокое погружение

ВМВ
Одесса
2007

ББК 32.973.2-018
М 642
УДК 004.438

Рецензент
Мазурок И.Е., кандидат технических наук, доцент кафедры математического обеспечения компьютерных систем Одесского национального университета им. И.И. Мечникова

Императивное и объектно-ориентированное программирование на Turbo Pascal и Delphi / А.С. Миронченко – Одесса: ВМВ, 2007. – 408 с.: ил.

ISBN 978-966-413-039-1

В книге последовательно излагаются концепции императивного и объектно-ориентированного программирования и реализация этих принципов в языках Turbo Pascal и Delphi. Анализируются недостатки объектно-ориентированного программирования и предлагается метод для их устранения.

Учебник подходит как для занятий в учебных заведениях, так и для самостоятельного изучения языков Turbo Pascal и Delphi. В учебнике более 140 примеров программ и около 210 задач по программированию и математике для самостоятельного решения.

Для студентов вузов, учащихся старших классов школ и лицеев с углубленным изучением информатики и физико-математических дисциплин, а также для всех интересующихся информатикой и программированием.

$М \frac{2404010000}{2007}$

© Миронченко А.С., 2007

Краткое содержание

Предисловие 10
Благодарности 11
Список условных обозначений 12
Часть 1: Императивное программирование 13
Глава 0. Программистское введение 13
Глава 1. Математическое введение 24
Глава 2. Первые шаги 41
Глава 3. Использование оператора if 52
Глава 4. Операторы циклов 66
Глава 5. Несколько операторов на закуску 81
Глава 6. Массивы 89
Глава 7. Подпрограммы 106
Проект 1. Длинная арифметика 126
Глава 8. Рекурсия 130
Глава 9. Модули и записи 154
Глава 10. Матрицы, множества и перечисления 165
Проект 2: Теория голосования 184
Глава 11: Символы и строки 189
Глава 12: Файлы 196
Проект 3: Эволюционно стабильные стратегии 207
Глава 13. Динамическая память 213
Проект 4. Поиск сходных слов 244
Глава 14. Выведение 247
Часть 2: Объектно-ориентированное программирование 256
Предисловие ко 2-й части 256
Глава 15. Основы Delphi 257
Глава 16. Объектно-ориентированное программирование 278
Проект 5: Криптография 320
Глава 17. Визуальное программирование 323
Глава 18. Графика 351
Проект 6: Беовульф и Нибелунги 368
Глава 19. Проблемы ООП. Везенспрограммирование 372
Напутствие 387
Решения и ответы 388
Список литературы 403

Полное содержание

Предисловие 10
Благодарности 11
Список условных обозначений 12
Часть 1: Императивное программирование 13
Глава 0. Программистское введение 13
0.1. Какие бывают языки программирования 13
0.2. Устройство ПК 18
0.3. Классификация языков программирования 20
0.4. Как ПК понимает языки программирования 22
Глава 1. Математическое введение 24
1.1. Множества 24
1.2. Операции над множествами 24
1.3. Числа 25
1.4. Системы счисления 30
1.5. Арифметические действия в двоичной системе счисления 31
1.6. Представление двоичных чисел в ПК 32
1.7. Шестнадцатеричная система счисления 34
1.8. Более общая точка зрения на системы счисления 34
1.9. Фибоначчиева система счисления 35
1.10. Метод математической индукции 37
1.11. Логарифмы 38
1.12. Последовательности и прогрессии 38
Задачи 39
Глава 2. Первые шаги 41
2.1. Базовые концепции процедурного программирования 41
2.2. Что нам понадобится для работы 43
2.3. Первые программы 43
2.4. Ввод значений с клавиатуры. Процедура Readln(x) 49
2.5. Несколько заключительных слов 50
Задачи 50
Глава 3. Использование оператора if 52
3.1. Немного справочной информации 52
3.2. Структура программы в Turbo Pascal 53
3.3. Числовые типы данных 54
3.4. Ограничены ли вычислительные возможности компьютера? ... 55
3.5. Условный оператор 56
3.6. Условный оператор с ветвью else 59
3.7. Делимость чисел. Операции div, mod 59
3.8. Решение линейного уравнения 60
3.9. Тип Boolean 62
3.10. Логические операции 62
Задачи 65
Глава 4. Операторы циклов 66
4.1. Цикл for 66
4.2. Вычисление факториала 67

4.3. Циклы while и repeat 69
4.4. Нахождение НОК и НОД 2-х чисел 70
4.5. Процедуры break и continue 74
4.6. Проверка чисел на простоту 75
4.7. Как компьютер может вычислить значение функции? 76
Задачи 78
Глава 5. Несколько операторов на закуску 81
5.1. Оператор case 81
5.2. Символьный тип 82
5.3. Работа с клавиатурой 83
5.4. Генератор псевдослучайных чисел 85
5.5. Директивы компилятора 86
5.6. Что мы уже знаем о структурном программировании 87
Глава 6. Массивы 89
6.1. Что такое массив? 89
6.2. Примеры работы с массивами 90
6.3. Моделирование многочленов 92
6.4. Поиск в массиве 96
6.5. Сортировка массива 98
6.6. Сортировка методом пузырька (обмена) 98
6.7. Сортировка выбором 100
6.8. Сортировка вставками 101
6.9. Сравнение простейших алгоритмов сортировки 102
Задачи 104
Глава 7. Подпрограммы 106
7.1. Описание подпрограмм 106
7.2. Передача массивов в подпрограммы 109
7.3. Параметры подпрограмм 111
7.4. Локальные и глобальные переменные 115
7.5. Открытые параметры-массивы 115
7.6. Совместимость и приведение типов 116
7.7. Бестиповые ссылки 118
7.8. Процедурный и функциональный типы 121
Задачи 124
Проект 1. Длинная арифметика 126
Глава 8. Рекурсия 130
8.1. Рекурсия в биологии 130
8.2. Рекурсия в литературе 130
8.3. Рекурсия в математике 131
8.4. Зачем нужна рекурсия в программировании? 132
8.5. Алгоритм Евклида 134
8.6. Вычисление натуральной степени числа 135
8.7. Приближенное решение уравнения методом бинарного деления 135
8.8. Рекурсия contra Итерация 138
8.9. Быстрая сортировка 141
8.10. Генерирование перестановок 143
8.11. Переборные алгоритмы 145

8.12. Косвенная рекурсия 147
8.13. В чем итерация выигрывает у рекурсии 150
8.14. Рекурсия в LISP 150
8.15. Сопрограммы 151
Задачи 152
Глава 9. Модули и записи 154
9.1. Модули и их структура 154
9.2. Взаимодействие модулей 158
9.3. Тип Запись 159
9.4. Оператор with 162
9.5. Записи с вариантами 162
Задачи 163
Глава 10. Матрицы, множества и перечисления 165
10.1. Матрицы 165
10.2. Обобщение процедур работы с матрицами 167
10.3. Матрицы в математике 170
10.4. Действия над матрицами 171
10.5. Множества 173
10.6. Сортировка подсчетом 174
10.7. Перечисляемый тип 176
10.8. Магические квадраты 178
Задачи 181
Проект 2: Теория голосования 184
Глава 11: Символы и строки 189
11.1. Массивы символов 189
11.2. Тип String 189
Задачи 195
Глава 12: Файлы 196
12.1. Логические и физические файлы 196
12.2. Открытие и закрытие файлов 196
12.3. Текстовые файлы 197
12.4. Типизированные файлы 198
12.5. Последовательный и прямой доступ к файлам 199
12.6. Буфер ввода/вывода 200
12.7. Бестиповые файлы 201
12.8. О количестве обращений к файлу 203
Задачи 206
Проект 3: Эволюционно стабильные стратегии 207
1. Постановка задачи 207
2. Моделирование жизни популяции 208
Глава 13. Динамическая память 213
13.1. Структура памяти, доступной программе 213
13.2. Использование бестиповых указателей 216
13.3. Указатель Nil 219
13.4. Обобщенная сортировка массива 220
13.5. Указатели и ссылки 223
13.6. Динамические структуры данных 223

13.7. Стеки 224
13.8. Однонаправленные списки 227
13.9. Кольцевые списки 230
13.10. Характеристика списков 233
13.11. Деревья 233
13.12. Разбор формулы 234
13.13. Сортировка с помощью дерева 239
13.14. Характеристика деревьев 242
Задачи 242
Проект 4. Поиск сходных слов 244
Глава 14. Выведение 247
14.1. Программа и модульное программирование 247
14.2. Простейший язык программирования 248
14.3. Все ли могут алгоритмы? 250
14.4. Простейший модульный язык программирования 252
14.5. Взаимодействие с операционной системой 252
14.6. Загрузка ОС 255
Часть 2: Объектно-ориентированное программирование 256
Предисловие ко 2-й части 256
Глава 15. Основы Delphi 257
15.1. Среда Delphi 257
15.2. Консольное приложение в Delphi 258
15.3. Изменения в Delphi (по сравнению с TP) 259
15.4. Работа с динамической памятью 265
15.5. Использование диалогов 269
15.6. Работа с файлами 270
15.7. Файлы, связанные с проектом 273
15.8. Динамически подключаемые библиотеки (DLL) 274
15.9. Подключение библиотеки 275
15.10. Использование модулей в DLL 277
15.11. Что нам дает DLL? 277
Глава 16. Объектно-ориентированное программирование 278
16.1. Модульный подход и ООП 278
16.2. Как объединить данные с подпрограммами средствами процедурного программирования 278
16.3. Объявление класса. Способы доступа к элементам класса 280
16.4. Конструкторы и деструкторы 281
16.5. Агрегация классов 283
16.6. Наследование 287
16.7. Виртуальные функции, полиморфизм и динамическое связывание 292
16.8. Смешанные иерархии 294
16.9. Иерархия классов Delphi 295
16.10. Абстрактные классы 296
16.11. Методы класса 298
16.12. Класс TObject 298
16.13. Приведение классов. Операторы is, as 299

16.14. Метаклассы 301
16.15. Методы метаклассов и метаобъектов 304
16.16. Свойства 305
16.17. Индексированные свойства 307
16.18. Взаимодействие классов 309
16.19. Интерфейсы 310
16.20. Новое в Delphi 2005 314
Задачи 319
Проект 5: Криптография 320
Глава 17. Визуальное программирование 323
17.1. Оконные приложения в Windows 323
17.2. Создание оконных приложений в Delphi 324
17.3. Первое оконное приложение 324
17.4. Добавляем компоненты на форму 326
17.5. Убегающая кнопка. Обработчик события OnMouseMove 328
17.6. Установка пароля на программу 332
17.7. Динамическое создание компонентов 332
17.8. Общие свойства компонентов 333
17.9. Свойства элементов управления (TControl) 334
17.10. События мыши и клавиатуры 334
17.11. TComboBox 335
17.12. Описание работы калькулятора 335
17.13. Исключения 340
17.14. Класс TList 344
17.15. Процессы и потоки 346
Задачи 350
Глава 18. Графика 351
18.1. Цвет кисти и пера 351
18.2. Рисование многоугольников 351
18.3. Рисование линий 352
18.4. Рисуем шашечную доску 354
18.5. Фракталы 357
18.6. Построение самоподобных фракталов 361
18.7. Реализация черепашьей графики 362
Задачи 366
Проект 6: Беовульф и Нибелунги 368
Глава 19. Проблемы ООП. Везенспрограммирование 372
19.1. История языков программирования 372
19.2. Обзор некоторых объектно-ориентированных языков 373
Simula 373
Smalltalk 373
Delphi (Object Pascal) 374
C++ 375
Java 378
19.3. Компиляция или интерпретация 379
19.4. Недостатки объектно-ориентированных языков 380
Описание самоорганизующихся систем 380

Объекты языкового уровня и объекты времени выполнения . 380
Взаимодействие программы с операционной системой 381
19.5. Несколько терминов из биологии 381
19.6. Везенспрограммирование (Wesensprogrammierung) 382
Основные понятия 382
Жизнь существа 383
Убираем ненужные технические понятия 384
Описание программ на разных языках 385
Общая картина 386
19.7. Что мы с вами еще не сделали 386
Напутствие 387
Решения и ответы 388
Глава 1 388
Глава 2 390
Глава 3 391
Глава 4 392
Глава 6 396
Глава 7 397
Глава 8 399
Глава 10 400
Глава 11 400
Глава 18 401
Список литературы 403

Предисловие

Мне ненавистно все, что только поучает меня, не расширяя и непосредственно не оживляя моей деятельности.

Иоганн Вольфганг Гёте

В книге рассматриваются концепции императивного и объектно-ориентированного программирования и реализация этих принципов в языках Turbo Pascal и Delphi. Кроме того, в последней главе автор анализирует недостатки современных языков программирования и предлагает собственную концепцию везенспрограммирования, которая должна устранить многие из проблем.

Книга разделена на 2 части. В первой из них (главы 0-14) рассматриваются концепции императивного программирования.

В главах 0,1 вводятся основные понятия программирования и вкратце рассматриваются концепции языков программирования, существующие на сегодняшний день.

В главах 2-13 рассматривается язык Turbo Pascal, один из наиболее популярных языков программирования. По мере роста знаний читателя изученные понятия рассматриваются с более общих позиций, что позволяет глубже понять материал.

Глава 14 - заключительная в первой части учебника. В ней рассматривается ряд общих вопросов, связанных с процедурным программированием (написание обобщенных программ, структурированность языков) и с программированием вообще (понятие машин Тьюринга, ограниченность алгоритмических машин).

Вторая часть (главы 15-19) посвящена объектно-ориентированному программированию (ООП) и взаимодействию программ с операционной системой.

В 19-й главе, после того, как вы уже основательно изучили императивные и объектно-ориентированные языки, автор анализирует достижения в области языков программирования и современное состояние языков. На мой взгляд, последние десятилетия не принесли для объектно-ориентированного программирования ничего существенного нового, а все изменения сводились к залатыванию дыр в существующих языках и обрастанию языков лишними понятиями. Поэтому автор предлагает свой выход из создавшегося положения – в переходе к **везенспрограммированию.**

Книга содержит более 140 примеров с исходным кодом (около 7500 строк кода), в которых рассматриваются многие важные алгоритмы, часто встречающиеся на практике. В учебнике есть около 210 задач по программированию и математике, которые можно разделить на 3 уровня:

1. Задачи разной сложности, позволяющие лучше усвоить пройденный материал.
2. Задачи, расширяющие материал книги. Выполняя их, вы сможете не только потренироваться в решении интересных задач, но и глубже понять рассмотренные в главах темы.
3. Проекты. В проектах вам предлагается тема для работы, а также несколько советов по ее выполнению. Работая над каким-то из проектов, вы можете посмотреть, на что вы способны.

Миронченко Андрей

Благодарности

Прежде всего я благодарю моих родителей за терпение и уважение к моему труду, за справедливую критику и за дизайн обложки. Благодарю доцента кафедры математического обеспечения компьютерных систем ОНУ им. И.И. Мечникова Мазурка Игоря Евгеньевича, который прочитал рукопись книги и сделал ряд важных замечаний по её тексту. Его работа позволила улучшить изложение некоторых тем. Неоценимую помощь оказал мой друг Владимир Сергиевский, который дважды прочитал рукопись книги и дал множество советов, улучшивших содержание многих глав. Обмен мнениями с Володей позволил избавиться от однобокости освещения некоторых разделов.

Список условных обозначений

$x \in A$	элемент x принадлежит множеству A
$x \notin A$	элемент x не принадлежит множеству A
$A \bigcup B$	объединение множеств A и B
$A \bigcap B$	пересечение множеств A и B
$A \subseteq B$	A является подмножеством B
A / B	разность множеств A и B
$a \vdots b$	a делится на b
$a \not\vdots b$	a не делится на b
$A \Leftrightarrow B$	выражение A верно тогда и только тогда, когда верно выражение B
$A \Rightarrow B$	из A следует B
N	множество натуральных чисел
Z	множество целых чисел
Q	множество рациональных чисел
R	множество вещественных чисел
C	множество комплексных чисел
(!)	задача, требующая знаний, выходящих за школьную программу
:	такое, что
$\exists$	существует (например: $\exists a \in Z$ существует целое число a).
$\forall$	для любого (например: $\forall x \in R \Rightarrow x^2 \geq 0$ для любого вещественного числа x следует, что $x^2 \geq 0$).
$i = \overline{1, n}$	i пробегает целые числа от 1 до n
$\sum_{i=k}^{l} a_i$	$a_k + a_{k+1} + \ldots + a_l$
$\prod_{i=k}^{l} a_i$	$a_k \cdot a_{k+1} \cdot \ldots \cdot a_l$

Часть I
Императивное программирование

Глава 0: Программистское введение

Первые программы мы начнем писать во второй главе. Но прежде мы должны ответить на ряд фундаментальных вопросов:

1. Что такое языки программирования и какими они бывают?
2. Как компьютер понимает, что написано в программном коде?
3. Где находится программа во время выполнения?

Введем несколько базовых определений:

- **Программирование** – написание программ.
- **Алгоритм** – упорядоченная последовательность действий.
- **Программный код** – запись алгоритма на каком-то языке (не обязательно языке программирования).

Под программой будем временно понимать последовательность действий, которые должен выполнить персональный компьютер (ПК), чтобы реализовать некоторый алгоритм. Более точное определение программы будет дано позже в этой главе.

Языки программирования нужны для того, чтобы передавать наши мысли компьютеру. Упрощенно можно сказать, что язык программирования – это набор правил, с помощью которых можно написать программный код.

0.1. Какие бывают языки программирования

Языков программирования – тысячи, но всех их можно разбить всего на несколько групп. Причем для этого совсем не надо быть великим программистом – вы сами сейчас легко сделаете это, даже если вы никогда не писали ни строчки программного кода. Представьте себе, что вы раскопали город неведомой доселе цивилизации, и разглядываете какие-то надписи, о языке и даже о предназначении которых вы не имеете ни малейшего представления – вы можете анализировать лишь внешний вид текста. Я предлагаю вам 7 программ на разных языках программирования. Вы должны просто пробежать по ним глазами и выделить сходства и различия.

Программа на C++

```
#include <iostream>
using namespace std;

void main()
{
     int n;
     cout<<"Enter quantity of numbers"<<endl;
     cin>>n;
```

```
     int F1=1;
     int F2=1;
     int F3;
     cout<<"The fibonacci's numbers"<<endl;
     cout<<1<<endl<<1<<endl;
     for (int i=1;i<=n-2;i++)
     {
          F3=F1+F2;
          F1=F2;
          F2=F3;
          cout<<F3<<endl;
     }
}
```

Программа на Turbo Pascal

```
var
  i,F1,F2,F3,n:integer;
begin
  writeln('Enter the quantity of numbers');
  readln(n);
  F1:=1;
  F2:=1;
  writeln('Fibonacci''s numbers');
  writeln(F1);
  writeln(F2);
  for i:=1 to n-2 do
    begin
    F3:=F1+F2;
    F1:=F2;
    F2:=F3;
    writeln(F3);
    end;
end.
```

Программа на машинном языке

```
  1  0000
  2  0000  40*(??)
  3  0040
  4
  5  0000
  6  0000  45 6E 74 65 72 20 51+
  7         75 61 6E 74 69 74 79+
  8         20 6F 66 20 6E 75 6D+
  9         62 65 72 73 0D 0A 24
 10  001C  46 69 62 6F 6E 61 63+
 11         63 69 20 6E 75 6D 62+
 12         65 72 73 0D 0A 24
 13  0030
 14
 15  0000
 16
```

```
17
18
19   0000
20   0000  B8 0000s
21   0003  8E D8
22
23   0005  B4 09
24   0007  BA 0000r
25   000A  CD 21
26
27   000C  E8 0000e
28   000F  8B C8
29   0011  B4 09
30   0013  BA 001Cr
31   0016  CD 21
32
33   0018  83 E9 02
34   001B  B8 0001
35   001E  E8 0000e
36   0021  B8 0001
37   0024  E8 0000e
38   0027  BB 0001
39   002A  BA 0001
40
41   002D  8B C2
42   002F  03 C3
43   0031  53 52 50
44   0034  E8 0000e
45   0037  58 5A 5B
46
47   003A  8B DA
48   003C  8B D0
49   003E  E2 ED
50
51   0040  B8 4C00
52   0043  CD 21
53   0045  CB
54   0046
55   0046
```

Программа на Prolog

```
predicates
 fibo(integer)
 fibo(integer,integer,integer)
clauses
 fibo(FN):-write(1),nl,FN>1,write(1),nl,FN>2,N=FN-2,fibo(1,1,N).
 fibo(_,_,0):-!.
 fibo(F1,F2,N):-F3=F1+F2,write(F3),nl,NN=N-1,fibo(F2,F3,NN).
goal write("Enter quantity of numbers"),nl,readint(X),
     write("Fibonacci's numbers"),nl,fibo(X).
```

Программа на Java

```
import javax.swing.JOptionPane;

public class Fibojava {
  public static void main (String args[]){
    String Numm;
    Numm=JOptionPane.showInputDialog("Enter quantity of numbers");
    int n=Integer.parseInt(Numm);
    int F1=1;
    int F2=1;
    int F3;
    System.out.println("The fibonacci's numbers");
    System.out.println(1+"\n"+1);
    for (int i=1;i<=n-2;i++)
    {
     F3=F1+F2;
     F1=F2;
     F2=F3;
     System.out.println(F3);
    }
  }
}
```

Программа на LISP

```
(defun Fibo (f1 f2 n)
(when (> n 0)
(print f1)
(Fibo f2 (+ f1 f2) (- n 1))
)
)
(Fibo 1 1 10)
```

Программа на Assembler

```
STK  SEGMENT        STACK
DB   64   DUP(?)
STK  ENDS

DATA SEGMENT
     MES  DB   'Enter Quantity of numbers',13,10,'$'
     MES2 DB   'Fibonacci numbers',13,10,'$'
DATA ENDS

CODE SEGMENT
ASSUME    CS:code, DS:data, SS:stk
  extrn read:near
  extrn write:near
START proc far
     mov ax,data
     mov ds, ax
```

```
        mov    ah,9
        mov    dx,OFFSET Mes
        int 21h

     call read
     mov cx,ax
        mov    ah,9
        mov    dx,OFFSET Mes2
        int 21h

     sub cx,2
     mov ax,1
     call write
     mov ax,1
     call write
     mov bx,1
     mov dx,1

fiba:    mov ax,dx
     add ax,bx
     push bx dx ax
     call write
     pop ax dx bx

     mov bx,dx
     mov dx,ax
     loop fiba

     mov ax,4c00h
     int 21h
     ret
start endp
CODE ENDS
END Start
```

Все программы делают одни и те же действия: печатают все числа Фибоначчи (эти числа описаны в главе 1) до номера, который вводит пользователь (либо до заранее заданного числа).

Большинство ребят, с которыми я занимался программированием, разбили языки на 5 групп:

1. Машинный язык
2. C++, Turbo Pascal, Java
3. Prolog
4. LISP
5. Assembler

Прежде, чем вкратце описать эти пять групп языков, мы должны познакомиться с устройством компьютера.

0.2. Устройство ПК

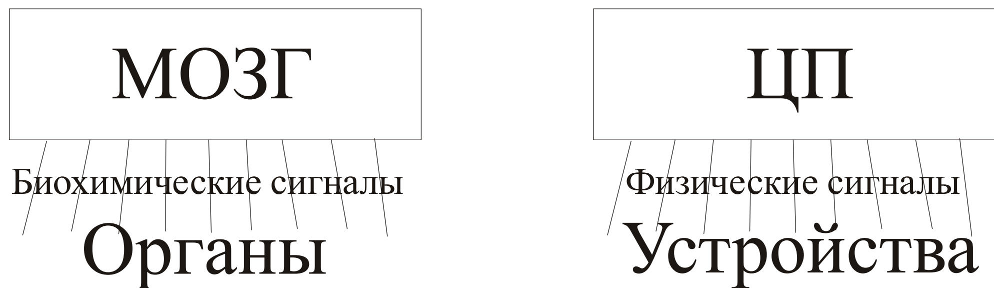


Рис 0.1: Сравнение работы организма животного и ПК

Давайте сравним ПК с организмом животного. Прежде, чем двинуть рукой, мозг вырабатывает множество нервных импульсов, а в кровь поступают определенные гормоны, которые дают указания клеткам тела, что и как им надо делать. То же самое происходит и в компьютере, только вместо биохимических процессов в его «теле» действуют физические: «мозг» компьютера, - его центральный процессор (ЦП), обменивается электрическими сигналами со своими «органами» (различными устройствами – монитором, принтером и т.д.), т.е. дает им некоторые команды.

Программа представляет собой последовательность элементарных (неделимых) команд процессора.

Нас будут интересовать прежде всего 2 «органа» компьютера: процессор и память.

- Память – устройство для хранения информации.

В современных компьютерах память состоит из набора битов.

- Бит – единица памяти, которая может находиться лишь в двух состояниях. Эти состояния мы будем в дальнейшем отождествлять с цифрами 0, 1.

Устройства, которые хранят 1 бит информации, могут быть различны. Например, оперативная память состоит из конденсаторов, которые могут находиться в состояниях заряжен/разряжен.

Для того чтобы можно было удобно работать с памятью, она является адресуемой.

- Минимальной адресуемой единицей памяти является **байт** (8 битов). У каждого байта есть свой номер и у разных байтов номера не могут совпадать.

Также используются следующие единицы количества информации:

1 Кб (килобайт) $= 2^{10}$ байт
1 Мб (мегабайт) $= 2^{10}$ Кб
1 Гб (гигабайт) $= 2^{10}$ Мб
1 Тб (терабайт) $= 2^{10}$ Гб

Память компьютера можно разбить на несколько уровней (см. рис. 0.2).

Скорость обмена информацией между процессором и памятью полностью регулируется физическими законами. Электрический сигнал, как и любой другой, не может двигаться быстрее скорости света (в вакууме $\approx 300000\frac{км}{с} = 3\cdot 10^{10}\frac{см}{с} = 30\frac{см}{нс}$), поэтому чем ближе память находится к ЦП, тем быстрее он будет с нею работать.

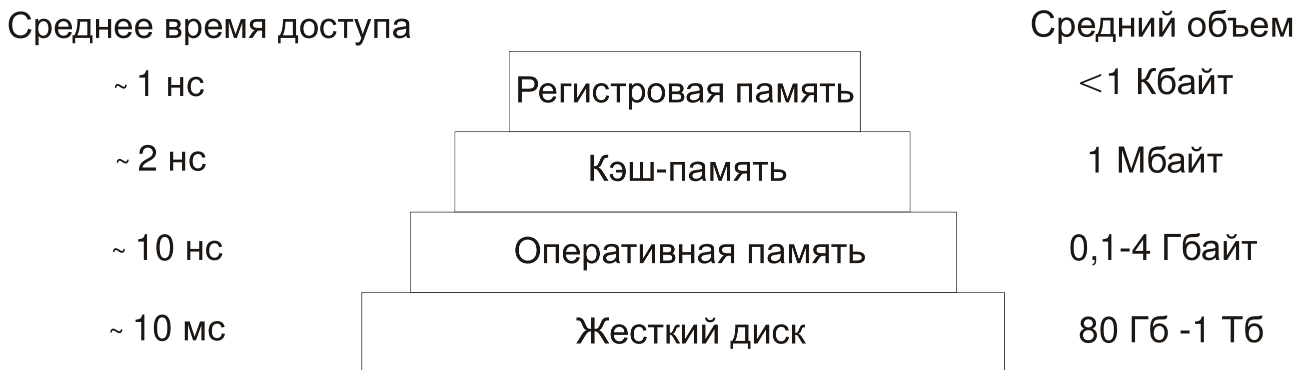


Рис 0.2 Структура памяти компьютера

Регистровая память находится в самом процессоре, поэтому доступ к ней наиболее быстрый. Кэш находится в непосредственной близости к процессору, поэтому обмен информацией также осуществляется довольно быстро. Оперативная память (ОП) находится дальше кэша, поэтому работа с ней ведется еще медленнее. То, что работа с жестким диском (ЖД) ведется намного медленнее, чем с ОП, зависит не от того, где он находится, а от структуры ЖД.

Процессор при работе с командами похож на хозяйку во время приготовления пищи: всю работу она проделывает на небольшой доске, несколько важных ингредиентов держит на столе, в случае надобности берет продукты из холодильника и лишь в крайнем случае идет в кладовку. Процессор действует аналогично: вся непосредственная работа идет в регистрах. Наиболее важные и часто используемые команды держатся в кэше, основной блок – в ОП, а наиболее объемные данные находятся на жестком диске.

Значения, которые были записаны в регистрах, кэше и ОП при выключении компьютера стираются (вследствие технического строения этой памяти). Поэтому без жестких дисков, Flash-памяти или CD/DVD или других носителей информации, на которых можно записывать данные на долгое время, обойтись нельзя. Но почему не убрать, скажем, ОП, и вместо этого сделать огромный кэш? Ответ прост: 1 Мб кэша стоит на порядок дороже, чем 1 Мб ОП. Регистровая память стоит примерно в 10 раз дороже, чем кэш. Именно поэтому придерживаются такой многоуровневой структуры памяти (сейчас кэш тоже состоит из нескольких уровней, причем чем дальше уровень от ЦП, тем больше он по объему памяти и дешевле по стоимости).

Работает процессор так: он считывает команду из кэша, выполняет ее, устанавливает новые значения регистров, считывает новую команду и т.д. Важно в этой схеме то, что процессор в единицу времени может обрабатывать лишь одну команду. Следовательно, процессор (одноядерный) принципиально не может выполнять одновременно несколько программ. Для того чтобы создать имитацию параллельного выполнения нескольких программ, надо заставлять процессор поочередно выполнять команды то одной, то другой программы. Если делать это достаточно часто, то можно добиться желаемого эффекта одновременности работы приложений.

В первых компьютерах в оперативной памяти хранились только данные. Хранить в ОП программу первым предложил или Джон фон Нейман, или Дж. П. Эккерт-младший (кто именно, точно неизвестно). Для того чтобы программы можно было хранить в ОП, надо ПК разрабатывать так, чтобы он мог распознавать определенные битовые комбинации как представление соответствующих команд.

- Набор операций вместе с системой их кодирования называется **машинным языком**.

Машинный язык является неотъемлемой частью процессора, поэтому процессор может распознать и выполнить любую команду, записанную на машинном языке. Теперь мы можем ввести важнейшее определение:

- Программа – последовательность элементарных (неделимых) команд на машинном языке.

0.3. Классификация языков программирования

На заре развития компьютерных технологий программы писались только на машинном языке. Вы видели, что внешне такая программа представляет собой набор чисел, записанных в 16-ричной системе счисления (ее мы рассмотрим в главе 1). Разумеется, писать такие программы было очень утомительно, и заниматься этим могли лишь большие энтузиасты.

Программисты быстро решили, что так больше дело продолжаться не может, и им пришла в голову такая мысль: придумать названия для регистров и машинных команд, и записывать программу уже с учетом этих обозначений. Однако ПК не полиглот – он понимает только машинный язык, поэтому для перевода программного кода на машинный язык надо написать специальную программу. Так и сделали, а сами программы-переводчики назвали ассемблерами (т.е. сборщиками). Позднее подобные языки программирования тоже стали называть ассемблерами.

Стиль программирования на ассемблере тот же, что и на машинном языке. Просто стало легче читать и отлаживать программу. За разработку языков программирования, значительно более близких к естественному языку (процедурных языков) первой взялась Грейс Хоппер. Программист, пишущий программы на процедурном языке, тратил значительно меньше времени на отладку программы, т.к. значительную часть чисто технических проблем берет на себя компьютер. Например, человеку все равно, складывать вещественные числа или целые, а вот в ПК есть 2 команды: целочисленного сложения и сложения с плавающей точкой, т.к. способы хранения вещественных и целых чисел различны. В процедурных языках сложение вещественных и целых чисел выполняется однотипно, что для программиста очень удобно. Стиль написания программ на процедурных языках - алгоритмический, а не технический. В частности, в процедурных языках появились типы данных, например: целые числа, вещественные числа, строки. В ассемблерах типов данных не было: в памяти компьютера вся информация хранится в двоичном коде – одна и та же последовательность байтов может означать и рисунок, и набор чисел, и некоторую строку символов, и программист, который пишет на ассемблере, сам решает, как ему интерпретировать информацию.

Процедурные языки постепенно развились в модульные языки. В модульных языках можно разбивать программный код на несколько файлов, каждый из которых представляет собой коллекцию подпрограмм. Модуль обладает определенной самостоятельностью: у него есть собственные подпрограммы, переменные, константы, которые не видны вне модуля. Основная программа может подключать модули и использовать подпрограммы, содержащиеся в них. При этом упрощается разработка больших программ, т.к. одни и те же модули могут использоваться в различных приложениях.

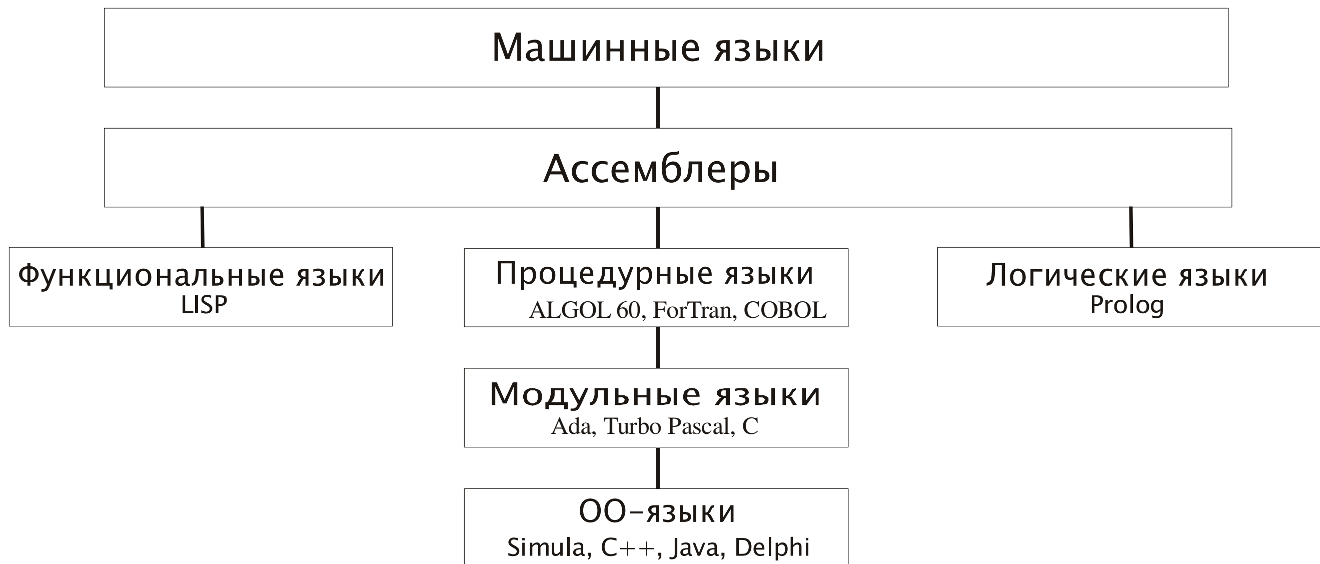


Рис 0.3 Классификация языков программирования

Следующим шагом стал переход к объектно-ориентированному программированию (ООП). Предположим, что нам надо смоделировать на компьютере работу часов. У часов есть детали (циферблат, стрелки и т.д.) и функции (показывать время, возможность выставления даты и т.д.). С точки зрения процедурного подхода детали можно объединить в единое целое, однако функции должны быть отделены от данных, а сгруппировать по смыслу их можно, используя модули. А согласно ООП детали и функции часов неотъемлемы друг от друга и составляют единый объект. В данном случае такое объединение действительно логично, однако, несмотря на все удобства, ООП иногда оказывается не в ладах со здравым смыслом, например: число пишет само себя, последовательность данных сама себя сортирует, а квадрат является более абстрактным объектом, чем прямоугольник, хотя ясно, что квадрат – частный случай прямоугольника.

Альтернативой процедурным языкам и их расширениям являются логические и функциональные языки. Логические языки (их еще называют декларативными) базируются на математическом аппарате формальной логики, и очень удобны для проведения логических рассуждений, поэтому они очень часто используется разработчиками искусственного интеллекта. Функциональные языки основываются на понятии функции. Но это понятие используется и в процедурных языках, поэтому мы отложим рассмотрение отличий между функциональными языками и процедурными до 8-й главы.

Функциональные, логические, процедурные языки называют языками высокого уровня (ЯВУ), в отличие от машинных языков и ассемблеров, которых называют языками низкого уровня.

Итак, мы с вами разбили все языки программирования на несколько классов, внутри каждого из которых принципиальных отличий нет. Хорошо изучив TP и Delphi, вы легко изучите C, C++, Java и подобные языки. Привыкать мыслить по-иному вам придется лишь при изучении языков других групп – ассемблеров, логических и функциональных языков.

Честно говоря, я умолчал о ряде гибридных языков, которые позволяют писать в разных стилях, кроме того, часто в ЯВУ можно писать и на ассемблере, если вам захочется.

В этой книге мы изучаем процедурные языки и их расширения, поэтому для того, чтобы вы лучше прочувствовали их сходства между собой, я приведу реализацию сортировки массива на C++ и TP (естественно, вы снова должны лишь посмотреть, не особенно вникая в содержимое).

Программа на C++

```
#include <iostream>
using namespace std;

#include <ctime>

void main()
{
  const n=9;
  int A[n];
  int i;
  srand(time(0));
  for (i=0;i<=n-1;i++)
    A[i]=rand()%100;

  cout<<"Initial massive"<<endl;
  for (i=0;i<=n-1;i++)
    cout<<A[i]<<' ';
  cout<<endl;

  for (i=n-2;i>=0;i--)
    for (int j=0;j<=i;j++)
      if (A[j]>A[j+1])
      {
         int tmp=A[j];
         A[j]=A[j+1];
         A[j+1]=tmp;
      }
  cout<<"Sorted array"<<endl;
  for (i=0;i<=n-1;i++)
       cout<<A[i]<<' ';
}
```

Программа на TP

```
const
  n=9;
var
  A:array [0..n-1] of integer;
  i,j,tmp:integer;
begin
  randomize;
  for i:=0 to n-1 do
    A[i]:=random(100);

  writeln('Initial massive');
  for i:=0 to n-1 do
    write(A[i], ' ');
  writeln;

  for i:=n-2 downto 1 do
    for j:=0 to i do
      if (A[j]>A[j+1]) then
        begin
        tmp:=A[j];
        A[j]:=A[j+1];
        A[j+1]:=tmp;
        end;
  writeln('Sorted array');
  for i:=0 to n-1 do
    write(A[i], ' ');
end.
```

0.4. Как ПК понимает языки программирования

Так как компьютер понимает лишь язык машинных кодов, то программный код, написанный не на машинном языке, должен быть на него переведен. Вы помните, что для того, чтобы перевести код из ассемблера на машинный язык писалась специальная программа, которая также называлась ассемблером (assembler – сборщик). Та же ситуация и с языками высокого уровня – код, написанный на ЯВУ, тоже надо переводить на машинный язык (или сначала на ассемблер, а потом – на машинный язык). Для этого пишется программа, называемая **транслятором** (translator – переводчик).

Трансляция программы состоит из 3 этапов:

1. **Лексический анализ** – разбиение программного кода на самостоятельные единицы текста (эквивалентно разбиению рассказа на слова и знаки препинания).
2. **Синтаксический анализ** – распознавание структуры программы и роли отдельных ее частей (понимание смысла рассказа)
3. **Генерация кода** – перевод действий, «осмысленных» синтаксическим анализатором на машинный язык (или ассемблер).

После перевода программы с языка высокого уровня на машинный, остается еще ряд проблем: программа может состоять из нескольких исполняемых модулей (каждый из которых транслируется отдельно), поэтому надо связать все модули воедино и получить загрузочный файл – т.е. файл, который можно запускать на выполнение.

Кроме трансляторов часто используются **интерпретаторы** – программы, которые не создают запускающий файл, а сразу после идентификации команды выполняют ее и переходят к следующей команде. Трансляторы должны объединять (англ. compile) несколько машинных команд в группы, чтобы имитировать выполнение одного оператора языка высокого уровня, поэтому трансляторы часто называют компиляторами.

Для одного и того же языка программирования может существовать много различных программ, которые переводят программный код на нем на машинный язык, и, естественно, они могут генерировать различный исполняемый код. Вообще говоря, компиляторы могут генерировать более быструю последовательность команд, чем интерпретаторы, т.к. они могут анализировать больший объем кода и принимать более эффективные решения по оптимизации работы программы. Кроме того, если вы хотите откомпилировать программу на некотором языке, то после этого ее можно использовать на любой машине, и сама программа-компилятор при этом не нужна. А если предполагается, что программный код должен быть интерпретируемым, то на любом компьютере должен стоять интерпретатор этого программного кода.

Глава 1: Математическое введение

В дальнейшем нам понадобится ряд понятий из математики для того чтобы хорошо разобраться в программировании.

Если вам не терпится поскорее начать программировать, то прочитайте разделы, посвященные множествам, натуральным, целым и рациональным числам, а также системам счисления, - эти знания необходимы для того, чтобы понять материал первых глав, и переходите ко второй главе. Остальные темы можете читать по мере надобности. Однако помните, что их знание необходимо для понимания программирования.

1.1. Множества

Точного определения множества в математике не существует, да и существовать не может. Строгость математики начинается с того момента, когда выбрано поле действия, - некоторый набор аксиом и базовых понятий, на основе которых может строиться математическая теория.

Давайте теперь попытаемся дать определение множества, исходя непосредственно из нашего опыта:

- Множество – набор четко различимых элементов.

Описывать множества можно следующими способами:

1. Перечислением его элементов.
2. С помощью правила, генерирующего все элементы множества.

Например, следующие 2 определения задают множество, состоящее из чисел 1, 3, 5, 7:

$$A = \{1, 3, 5, 7\}$$

$$A = \{n \in N \mid n \not\vdots 2, n \leq 7\}$$

Вторую запись следует расшифровывать так: множество А состоит из натуральных чисел, которые не делятся на 2 и не больше 7.

То, что число x принадлежит множеству A, записывается так: $x \in A$.

- Множество А называется подмножеством множества В, если все элементы множества А принадлежат множеству В. Обозначается это: $A \subseteq B$

Множества А и В считаются равными, если они состоят из одинаковых элементов. Иначе это можно сказать так: $A = B$ тогда и только тогда, когда $A \subseteq B$ и $B \subseteq A$.

1.2. Операции над множествами

- Пересечением множеств А и В называется множество, состоящее из тех и только тех элементов, которые входят во множества А и В.

Обозначается пересечение множеств $A \bigcap B$.

- Объединением множеств А и В называется множество, состоящее из тех и только тех элементов, которые входят хотя бы в одно из множеств А и В.

Обозначается объединение множеств $A \bigcup B$.

- Разностью множеств А и В называется множество, состоящее из тех и только тех элементов, которые входят в множество А и не входят во множество В.

Обозначается разность множеств $A \setminus B$.

Например: если $A = \{1,3,5,7\}$, $B = \{1,5,6,8,7,10\}$, то
$A \bigcup B = \{1,5,7,3,6,8,10\}$
$A \bigcap B = \{1,5,7\}$
$A \setminus B = \{3\}$
$B \setminus A = \{6,8,10\}$

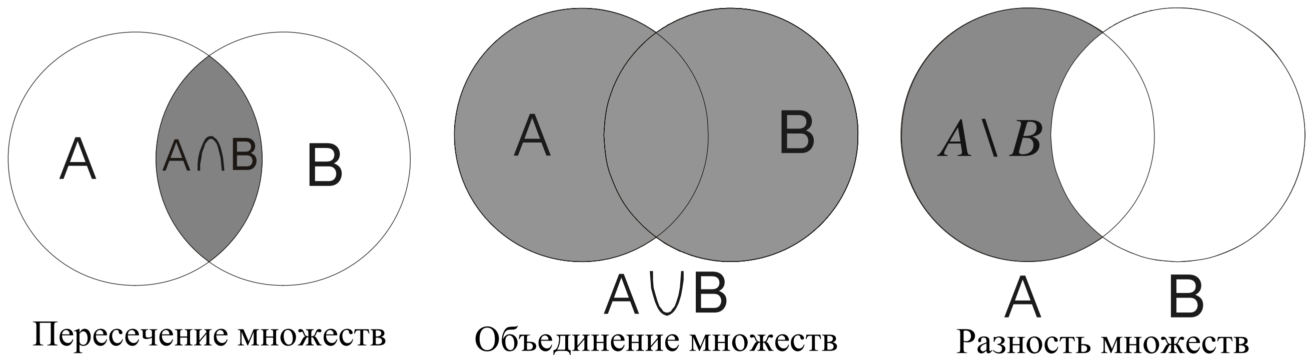


Рис 1.1 Операции над множествами

1.3. Числа

Теперь давайте разберемся с тем, что же такое число.

Вспомните, как учат считать детей: им дают несколько предметов, и называют их число. Потом дают другое множество предметов и снова называют их число и т.д. Это наталкивает на мысль о связи понятия числа с понятием множества. Если множество пустое (т.е. не содержит элементов), то его число элементов =0. Добавляя к пустому множеству любой другой элемент, мы получим новое множество, количество элементов которого =1. Добавляя к любому множеству, число элементов которого равно 1, еще один элемент, получим множество с количеством элементов, равное 2 и т.д. Фактически мы определяем число как характеристику, общую для некоторого класса множеств. Таким способом можно определить числа 0, 1, 2, 3, 4, … , которые называются **натуральными числами**[1].

Но натуральных чисел очень мало: даже простое уравнение $x+1=0$ в натуральных числах не имеет решений.

А уравнения подобного рода то и дело возникают на практике: если у вас не только нет денег, но вы еще и должны некоторую сумму, то выходит, что количество денег, имеющихся у вас, меньше 0. Чтобы решать подобные задачи, были введены **целые числа**: …, –3, -2, -1, 0, 1, 2, 3, … (натуральные числа и числа, обратные натуральным). Обозначается множество натуральных чисел буквой N, а множество целых – буквой Z.

Но и целых чисел немного: когда вам надо разделить торт на несколько человек, то вам приходится иметь дело с дробными числами. Множество **рациональных чисел** Q – множество чисел вида $\frac{m}{n}$, где m – целое число, а n – натуральное.

Например: $\frac{3}{2}, \frac{-3}{2}$, 1 - рациональные числа.

---

[1] В некоторых разделах математики 0 не считают натуральным числом, но в вычислительной математике 0 – натуральное число.

Часто надо представлять рациональные числа в виде десятичной дроби. Вопросы существования разложения решает следующая теорема:

Критерий рациональности: $q \in Q \Leftrightarrow q$ **- периодическая десятичная дробь**

Доказательство:

То, что любая конечная дробь – рациональное число следует из равенства:

$$x + 0.z_1 z_2 ... z_k = \frac{10^k x + z_1 z_2 ... z_k}{10^k}, \text{ где } z_i \text{ - цифры, } i = \overline{1, n}.$$

Докажем, что периодическая часть $0.(d_1 d_2 ... d_n)$, где все d_i это цифры – тоже рациональное число. Пусть $y = 0.(d_1 d_2 ... d_n)$. Тогда $10^n y = d_1 d_2 ... d_n + 0.(d_1 d_2 ... d_n) = d_1 d_2 ... d_n + y$. А отсюда следует, что:

$$y = \frac{d_1 d_2 ... d_n}{10^n - 1}$$

Т.к. любая периодическая дробь – это сумма конечной и периодической частей, а сумма рациональных чисел – рациональное число, то мы доказали, что любая периодическая дробь – рациональное число.

Теперь давайте докажем обратное: что любая несократимая дробь вида $\frac{m}{n}$ представима в виде десятичной дроби.

Будем делить m на n с остатком: $m = a_1 n + r_1$, где $0 \le r_1 < n$.

Если $r_1 = 0$, то $\frac{m}{n}$ - целое число. В противном случае $\frac{m}{n} = a_1 + \frac{r_1}{n} = a_1 + \frac{1}{10}\left(\frac{10 r_1}{n}\right)$.

Обозначим $m_1 = 10 r_1$, и разделим это число с остатком на n: $m_1 = a_2 n + r_2$, где $0 \le r_2 < n$.

Если $r_2 = 0$, то $\frac{m}{n}$ - целое число. В противном случае $\frac{m_1}{n} = a_2 + \frac{r_2}{n}$, причем $0 \le a_2 \le 9$. При этом

$$\frac{m}{n} = a_1 + \frac{r_1}{n} = a_1 + \frac{1}{10}\left(\frac{m_1}{n}\right) = a_1 + \frac{1}{10}\left(a_2 + \frac{r_2}{n}\right) = a_1 + \frac{a_2}{10} + \frac{1}{100}\left(\frac{10 r_2}{n}\right)$$

Значит a_2 - первая цифра в дробной части числа $\frac{m}{n}$. Если $r_1 = r_2$, то $m_1 = m_2$, и в дальнейшем десятичные знаки будут повторяться: $a_2 = a_3 = ... = a_k = ...$ и дробь будет периодической.

Далее, проводя аналогичные действия, получим:

$m_2 = a_3 n + r_3$,

…

$m_k = a_{k+1} n + r_{k+1}$

Если на каком-то из шагов мы получим остаток r_i, равный одному из остатков r_j, $j < i$, то: $m_i = m_j$ и $a_{i+s} = a_{j+s}$ для любого s, т.е. дробь будет периодическая, и ее периодом будет $a_{j+1} a_{j+2} ... a_i$.

Но мы знаем, что $0 \le r_i < n$, поэтому либо мы получим на каком-то шаге остаток равный 0, и дробь будет содержать конечное количество десятичных знаков, либо на каком-либо шаге мы получим остаток, который уже встречался ранее и дробь будет периодической. Теорема доказана.

Замечание: из доказательства следует, что период дроби $\frac{m}{n}$ не может содержать больше, чем n знаков.

Рациональных чисел было достаточно для решения большинства простых арифметических задач. Но давайте рассмотрим простейшую геометрическую задачу: найти гипотенузу прямоугольного треугольника с катетами равными 1. Для этого надо, согласно теореме Пифагора, решить уравнение $x^2 = 2$. Для того, чтобы записать решение этого уравнения, надо было ввести дополнительную алгебраическую операцию – извлечение квадратного корня. Решение уравнения тогда можно записать в виде $x = \pm\sqrt{2}$. Так как длина отрицательной быть не может, то искомая гипотенуза = $\sqrt{2}$. Тут стал вопрос: а является ли число $\sqrt{2}$ рациональным? Ответ на этот вопрос отрицательный; дал его сам Пифагор, причем результат очень опечалил его, так как он разбивал всю пифагорейскую философию.

Доказать, что $\sqrt{2}$ - не рациональное число можно, например, так:

Предположим противное: пусть $\sqrt{2} \in Q$. Тогда оно представимо в виде $\sqrt{2} = \frac{m}{n}$, причем дробь несократима. В таком случае $m^2 = 2n^2$, т.е. число m^2 - четное, а значит, четным будет и само число m. Пусть $m = 2k$, - тогда можно записать, что $(2k)^2 = 2n^2$, или $2k^2 = n^2$. Но в таком случае n - также четное, а значит дробь $\frac{m}{n}$ сократима, что противоречит нашему предположению. Поэтому число $\sqrt{2}$ - не рационально.

Вы видите, что указать, является ли рациональное число целым, легко даже по внешнему виду, а вот доказать рациональность числа – значительно сложнее. Но главное не это: основная проблема в том, каким образом надо построить новое числовое множество, которое бы включало в себя все рациональные числа, а также целую охапку новых чисел, одно из которых мы получили только что. И какими свойствами должно обладать новое множество?

Не будем вдаваться в детали построения множества действительных чисел, а ограничимся лишь общей идеей. Есть несколько методов построения действительных чисел. 3 наиболее известных из них принадлежат Георгу Кантору, Рихарду Дедекинду и Карлу Вейерштрассу. Мы будем следовать методу Дедекинда.

Будем считать, что рациональные числа уже построены и логически обоснованы. Будем рассматривать разбиения рациональных чисел на 2 подмножества C_1 и C_2 так, чтобы :

1. $C_1 \cup C_2 = Q$.
2. Для любых чисел $x \in C_1$, $y \in C_2$ следует, что $x < y$.

Для множеств C_1, C_2 справедливо одно из следующих соотношений:

1. В C_1 есть наибольший элемент, а в C_2 нет наименьшего элемента.
2. В C_1 нет наибольшего элемента, а в C_2 есть наименьший элемент.
3. В C_1 нет наибольшего элемента, а в C_2 нет наименьшего элемента.

Очевидно, что не может существовать наименьшего числа в C_2 и одновременно наибольшего в C_1: пусть эти числа соответственно равны x, y, тогда рациональным числом будет $\frac{x+y}{2}$, которое не будет принадлежать ни C_1, ни C_2, что противоречит тому, что $C_1 \cup C_2 = Q$.

В случаях 1, 2 сечение однозначно определяет рациональное число, равное наибольшему элементу C_1 в случае 1, и наименьшему элементу C_2, в случае 2.

В третьем случае мы будем говорить, что сечение определяет иррациональное число (это число больше любого числа из C_1 и меньше любого числа из C_2).

Например, число $\sqrt{2}$ определяется сечением (C_1, C_2), где $C_1 = \{x : x^2 < 2\}$, $C_2 = \{x : x^2 > 2\}$.

Так Дедекинд построил вещественные числа и, кроме того, показал, что если начинать строить сечения для R, то новых чисел уже не найти, т.е. область вещественных чисел нерасширяема. Таким образом, на основе рациональных чисел построены и действительные.

Хотя мы построили множество действительных чисел, но было бы неплохо иметь в распоряжении рабочее определение действительных чисел. Оно будет таким:

Множество **действительных**, или **вещественных**, чисел (обозначается R (reele Zahlen), реже используется обозначение D) – множество чисел x вида $x = n + 0.d_1 d_2 \ldots$, где n – целое число, d_i- цифра, причем в разложении десятичной дроби в конце не должно быть бесконечной последовательности девяток.

Определение вещественного числа может показаться странным. Почему вдруг число $0.(9)$ не вещественно? Чем оно хуже остальных?

А дело вот в чем: $0.(9) = 1$. Принципиально то, что равенство полагается не в смысле предела, т.е. что последовательность чисел 0.9, 0.99, 0.999, … стремится к 1, а в смысле обычного числового равенства, такого же, как и $4 - 1 = 3$. Это легко проверить: Пусть $x = 0.(9)$. Тогда $10x = 9.(9)$, или $10x = 9 + x$, откуда $x = 1$.

Это равенство демонстрирует очень важное свойство действительных чисел: бесконечно малых действительных чисел не существует (равно как и бесконечно больших).

Дело в том, что вещественные числа подчиняются аксиоме Евклида: для любых чисел a и b, где $a \neq 0$, существует число n такое, что $an > b$.

А из этой аксиомы следует, что чисел, отличных от нуля и при этом не изменяющих своей величины при умножении на любое вещественное число, не существует!

Поэтому числа, у которых периодом является число 9, исключаются из рассмотрения (вспомните, что во множестве должны быть четко различимые элементы).

Как следует из критерия рациональности, любое иррациональное число представляется в виде непериодической десятичной дроби. Однако не следует считать, что иррациональность числа означает «хаотичность» расположения цифр в его дробной части. Например, число 0,123456789101112…, в котором выписаны подряд все натуральные числа, является иррациональным (см. упражнение 7).

Обычно в школах на действительных числах рассмотрение числовых множеств заканчивают. Но вспомните уроки физики: разве не часто там говорят о бесконечно малых приращениях, скоростях, расстояниях? А ведь только что мы отметили, что никаких бесконечно малых чисел среди действительных нет. Что же это такое: выходит, что нас все время обманывали, или … надо просто расширить множество действительных чисел.

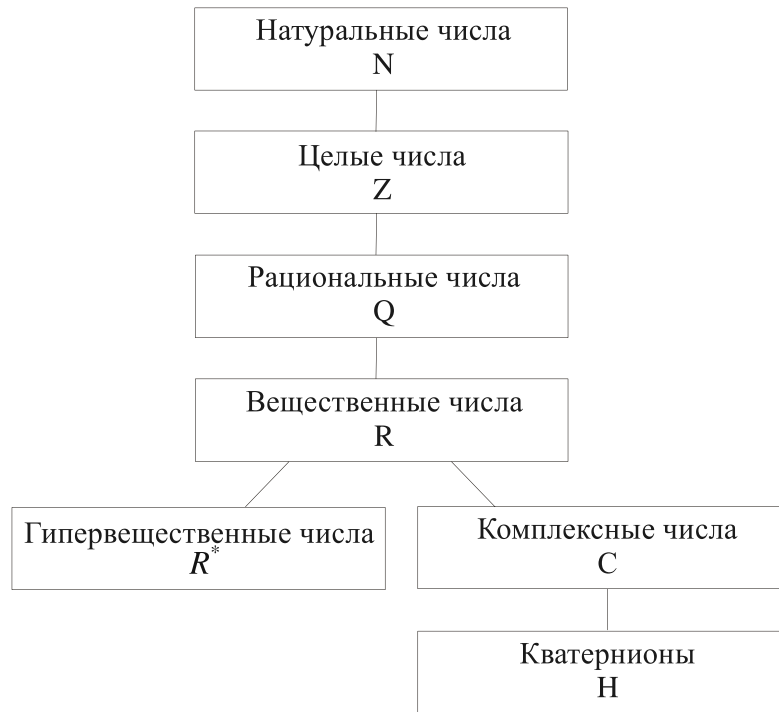


Рис 1.2 Основные числовые множества

Действовать будем огнем и мечом: если нам аксиома Евклида мешает использовать бесконечно малые и бесконечно большие числа, то давайте выбросим ее и дело с концом. В таком случае в числовую прямую будут втиснуты все числа вида $x+a$, где a- бесконечно малое число, а x - вещественное число. При этом под бесконечно малым числом понимается число, которое больше нуля, но при этом меньше любого положительного действительного числа.

Числа, полученные из вещественных отбрасыванием аксиомы Евклида, называются гипервещественными. Обозначается множество гипервещественных чисел R^* (не очень хорошее обозначение, т.к. его часто употребляют для обозначения множества действительных чисел без нуля).

Таким образом, при решении физических задач имеет смысл расширить множество чисел, чтобы выкладки стали более естественными[2].

Но можно расширять множество действительных чисел иначе. Когда мы строили множество вещественных чисел, мы обнаружили, что результат $x^2 = 2$ не будет рациональным числом. Давайте так же поступим и сейчас: попытаемся найти уравнение, которое не будет разрешимо в действительных числах. Долго думать не приходится: одним из таких уравнений является $x^2 = -2$. Нас учили в школе, что любое число, возведенное в квадрат, не может быть отрицательным. Но мы пропустим все запреты мимо ушей и формально вычислим квадратный корень: $x = \pm\sqrt{2} = \pm\sqrt{2}\sqrt{-1}$. Корень из –1 не вычисляется, поэтому для него вводится специальное обозначение: $i = \sqrt{-1}$. Тогда решения уравнения $x^2 = -2$ будут иметь вид $x = \pm\sqrt{2}i$.

Само же множество комплексных чисел C – это множество чисел вида $a+bi$, где a,b - вещественные числа.

[2] По правде говоря, физики не спешат использовать гипервещественные числа и не обращают внимания на корректность выкладок. Ведь для ученых-естественников критерий истины – практика.

Хорошо, скажете вы, мы придумали эти комплексные числа, но где же их применить? Все предыдущие множества чисел мы строили для решения конкретных задач.

На самом деле задач, в которых необходимо или, во всяком случае, удобно выйти в комплексную область в прикладных дисциплинах много[3]. Кроме того, как ни странно, но комплексные числа зачастую не усложняют, а наоборот, упрощают решение многих задач. Например: алгебраическое уравнение всегда имеет комплексный корень, а вот существуют ли среди них действительные корни - гораздо более сложный вопрос.

Вы видите, что множество действительных чисел нам удалось расширить двумя совершенно разными способами. До действительных чисел разветвлений не было, но не потому, что их нельзя придумать! Например, можно рассматривать множество чисел вида:

$Q_{\sqrt{2}} = \{x : x = a + b\sqrt{2},\ a, b \in Q\}$, которое также является вполне интересным расширением множества рациональных чисел. Значит вопрос о том, какие числа являются «правильными», не имеет смысла так же, как и стоявший ранее вопрос о том, какая же геометрия, евклидова или неевклидова, является корректной.

Справедливости ради следует отметить, что существуют и дальнейшие обобщения понятия числа, например, кватернионы, которые также имеют свои приложения к естественным наукам.

1.4. Системы счисления

Обычно мы используем позиционную десятичную систему счисления (ССч). Она называется десятичной потому, что количество цифр – десять: 0,1,…,9, а позиционной - так как значение цифры зависит от ее положения в записи числа. Само число 10 называется основанием десятичной ССч.

Если нам дано число, записанное в десятичной системе счисления, то нам, фактически, задается разложение этого числа по степеням десятки, где коэффициенты этого разложения – цифры 0, …, 9.

Например: $102 = 1 \cdot 10^2 + 0 \cdot 10^1 + 2 \cdot 10^0$.

Но что мешает нам разложить то же самое число по степеням не десятки, а какого-нибудь другого натурального числа, большего 1? Давайте, разложим число 102 по степеням двойки с коэффициентами равными 0 или 1.

$$102 = 64 + 32 + 4 + 2 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$$

Теперь запишем все коэффициенты в этом разложении подряд: 1100110. Мы с вами получили запись числа 102 в двоичной ССч, т.е. в ССч с основанием 2.

Записывается это так: $102_{10} = 1100110_2$. Индекс внизу означает основание ССч. В дальнейшем если индекса под числом нет, то будем считать, что оно записано в 10-й ССч.

Например, разложим число 102 по степеням 3-ки и 8-ки (с коэффициентами 0..2 и 0..7 соответственно):

$$102_{10} = 1 \cdot 3^4 + 2 \cdot 3^2 + 1 \cdot 3 = 1 \cdot 3^4 + 0 \cdot 3^3 + 2 \cdot 3^2 + 1 \cdot 3^1 + 0 \cdot 3^0 = 10210_3$$

$$102_{10} = 1 \cdot 8^2 + 4 \cdot 8^1 + 6 \cdot 8^0 = 146_8$$

Аналогичным образом можно переводить числа в любые ССч.

[3] Например, комплексные числа играют фундаментальную роль в квантовой механике.

Для закрепления материала я приведу еще несколько примеров представления чисел в двоичной системе:

$37 = 32 + 4 + 1 = 1 \cdot 2^5 + 1 \cdot 2^2 + 1 \cdot 2^0 = 100101_2$

$255 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 11111111_2$

Переводить числа из различных систем счисления назад в десятичную ССч еще проще. Например, для 2-й ССч формула перевода будет выглядеть так:

$(d_{k-1}d_{k-2}...d_0)_2 = 2^{k-1} \cdot d_{k-1} + 2^{k-2} \cdot d_{k-2} + ... + 2^0 \cdot d_0$, где все d_i - цифры 0 или 1. (1)

Теперь мы построим алгоритм перевода чисел из десятичной ССч в двоичную. Предположим сначала, что у нас есть некоторое неизвестное число в десятичной системе. Как бы мы искали его цифры. Цифру в младшем разряде легко найти, если вычислить остаток от деления числа на 10. Если же разделить его на 10 (с отбрасыванием остатка), то мы получим число, в котором отброшена последняя цифра. Например, если исходное число =124, то мы получим числа 4 и 12. Затем, применяя к усеченному числу те же действия, мы получим 2-ю цифру и т.д.

Фактически, для перевода из 10-й ССч в 2-ю мы будем использовать тот же самый алгоритм, только делить будем на двойку. К такому же алгоритму можно прийти и из других соображений: перепишем правую часть формулы (1) следующим образом:

$(d_{k-1}d_{k-2}...d_0)_2 = 2^{k-1} \cdot d_{k-1} + 2^{k-2} \cdot d_{k-2} + ... + 2^0 \cdot d_0 = d_0 + 2\Big(d_1 + 2\Big(d_2 + 2\big(d_3 + ... + 2\big(d_{k-2} + 2d_{k-1}\big)\big)\Big)\Big)$.

Если разделить слагаемое, стоящее в правой части, на 2, то в остатке будет d_0, а частным - $d_1 + 2\Big(d_2 + 2\big(d_3 + ... + 2\big(d_{k-2} + 2d_{k-1}\big)\big)\Big)$. Деля его на 2, получим остаток d_1, а частное - $d_2 + 2\big(d_3 + ... + 2\big(d_{k-2} + 2d_{k-1}\big)\big)$. И т.д.

Давайте рассмотрим пример: представим число 43 в двоичной ССч.

$43 = 1 + 2 \cdot 21$, поэтому $d_0 = 1$

$21 = 1 + 2 \cdot 10$, поэтому $d_1 = 1$

$10 = 0 + 2 \cdot 5$, поэтому $d_2 = 0$

$5 = 1 + 2 \cdot 2$, $\Rightarrow$ $d_3 = 1$

$2 = 0 + 2 \cdot 1$ $\Rightarrow d_4 = 0$

$1 = 1 + 2 \cdot 0$ $\Rightarrow$ $d_5 = 1$

Итого, получим: $43_{10} = 101011_2$

Ясно, что аналогичный алгоритм можно адаптировать и для перевода дробных, а не только натуральных чисел. Если дробь непериодическая, то проблем не возникает. В противном случае вышеприведенный алгоритм использовать нельзя, т.к. он будет бесконечен. Решение этой проблемы вынесено в упражнения.

1.5. Арифметические действия в двоичной системе счисления

Информация в современных компьютерах представляется в виде последовательности битов. Поэтому для ПК «родной» системой счисления является двоичная. Нашей первоочередной задачей будет научиться производить арифметические операции с натуральными числами, записанными в двоичной системе счисления.

Начнем со сложения. Таблица сложения для двоичных чисел предельно проста:

$0_2 + 0_2 = 0_2$

$1_2 + 0_2 = 0_2 + 1_2 = 1_2$

$1_2 + 1_2 = 10_2$

Складывать двоичные числа будем «в столбик», аналогично тому, как мы это делали в начальной школе с десятичными числами.

Рассмотрим пример 59 + 46 = 105

```
           1       1       1       1
 111011  111011  111011  111011  111011   111011
+101110 +101110 +101110 +101110 +101110  +101110
-------  ------  ------  ------  ------  -------
      1      01     001    1001   01001  1101001
```

Теперь давайте научимся умножать двоичные числа:

$0_2 \cdot 0_2 = 0_2 \qquad 1_2 \cdot 0_2 = 0_2 \cdot 1_2 = 0_2 \qquad 1_2 \cdot 1_2 = 1_2$

Лишь 4 простых равенства вместо 100, которые заставляют первоклашек заучивать наизусть! Вот простой пример умножения чисел в двоичной системе:

```
 . 110
    11
 -----
   110
 +110
 -----
 10010
```

Деление и вычитание в столбик, разумеется, также абсолютно аналогично делению и вычитанию в 10-й ССч.

Рациональные и иррациональные числа в 2-й ССч также вводятся очень просто, например,

$$1001.101_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 8 + 1 + \frac{1}{2} + \frac{1}{8} = 9\frac{5}{8}$$

1.6. Представление двоичных чисел в ПК

Любой переключатель может кодировать 2 значения: включен/выключен, которые можно отождествить с числами 0, 1. В компьютерной технике все аналогично: сделать память на основе элементов, которые могут находиться в 2-х состояниях проще, чем на основе элементов, принимающих 10 различных значений. Но просто перенести все рассуждения из математики в программирование нельзя. Причина лежит на поверхности: ведь кроме 0 и 1 мы использовали еще 2 символа: «-» и «.» (двоичную точку). Значит, перед тем, как использовать числа в ПК, надо избавиться от необходимости использовать эти 2 символа. Только представление натуральных чисел в математике и в ПК совпадают.

Представление целых чисел

Мы с вами рассмотрим лишь представление целых чисел. Пусть для определенности количество двоичных разрядов =3. Логичный способ такой: пусть старший бит числа будет знаковым: если он равен 0, то число – положительное, если 1 – то отрицательное. Т.е. все числа вида 0** - положительные, а вида 1** - отрицательные.

Теперь надо ввести порядок на множестве целых чисел. Это можно сделать различными способами. Естественно, лучшим способом будет такой, согласно которому арифметические операции и операции сравнения будут выглядеть наиболее просто.

Одним из таких «хороших» порядков будет следующий (для чисел, на которые выделяется 3 бита):

100	101	110	111	000	001	010	011
-4	-3	-2	-1	0	1	2	3

Называется такое представление двоичных чисел двоичным дополнительным кодом (ДДК). Неотрицательные числа, записанные в ДДК, в 10-ю систему переводятся так же, как и натуральные числа, записанные в обычной двоичной ССч. Процедура перевода отрицательных чисел будет иной: сначала инвертировать число (т.е. заменить нули на единицы, а единицы на нули), что соответствует взятию симметричного числа с другого края таблицы, а затем прибавить к полученному числу единицу (т.к. натуральные числа начинаются с 0, а не с 1), и результат перевести в десятичную систему как натуральное число. Ответом же будет это число со знаком «-».

Пример 1: пусть на запись целого числа выделяется 3 бита. Требуется перевести число $101_{ДДК}$ (число 101, записанное в ДДК) в десятичную систему счисления.

Решение: старший бит равен 1, поэтому число – отрицательное. Сначала инвертируем число 101. Получим 010. Прибавляя единицу, получим 011. Это число переводим в десятичную систему как обычное натуральное число: $011_2 = 3_{10}$.

Значит $101_{ДДК} = -3_{10}$.

Пример 2: пусть на запись целого числа выделяется 5 битов. Надо перевести число $10000_{ДДК}$ в 10-ю систему счисления.

Решение: старший бит равен 1, поэтому число – отрицательное. Проделаем необходимые операции: $10000 \to [\text{инвертируем}] \to 01111 \to [\text{прибавляем 1}] \to 10000$.

Теперь надо перевести число 10000 (это уже обычное двоичное натуральное число, а не целое число, записанное в ДДК).

$10000_2 = 16_2$

Значит $10000_{ДДК} = -16_{10}$.

Сравнивать положительные числа между собой можно просто: начать сравнение со старших битов и, как только будут найдены биты с различными значениями, то число, в котором находится 0, будет меньше. Видно, что абсолютно аналогично можно сравнивать и отрицательные числа, что очень хорошо. Более того, сложение и умножение чисел, заданных с помощью ДДК, можно выполнять по тому же алгоритму, что и умножение натуральных чисел.

Например:

$-2_{10} \cdot 2_{10} = 110_{ДДК} \cdot 010_{ДДК} = [\text{применяем обычный алгоритм}] = 1100$

Ответ в выражении занимает 3 бита, поэтому все биты, которые выходят за допустимые 3 бита, будут уничтожены, и результат получится: $110_{ДДК} \cdot 010_{ДДК} = 100_{ДДК} = -4_{10}$.

Чтобы понять, почему алгоритмы выполнения арифметических операций для чисел, записанных в ДДК, не отличаются от соответствующих алгоритмов для натуральных чисел, запишем целые числа, заданные в ДДК, так, как показано на рисунке 1.3.

Стрелка показывает направление роста двоичных чисел, которые интерпретируются как натуральные (при этом мы учитываем, что $111_2 + 1_2 = 1000_2 = [\text{отбрасываем старший бит}] = 0_2$). Прибавление единицы к некоторому числу означает сдвиг его по направлению стрелки. Прибавление –1 означает

прибавление $111_{ДДК}$, т.е. сдвиг на 7 единиц. Т.к. всего чисел $8=2^3$, то сдвиг на 7 единиц по направлению стрелки это то же самое, что и сдвиг на 1 единицу в обратную сторону. Абсолютно аналогично, прибавление $-2=110_{ДДК}$ означает сдвиг на 6 клеток по направлению стрелки = сдвиг на 2 клетки против движения стрелки. Следовательно, алгоритм сложения целых чисел, записанных в ДДК, точно такой же, как и алгоритм сложения обычных натуральных чисел. Аналогично можно рассмотреть и умножение целых чисел, записанных в ДДК.

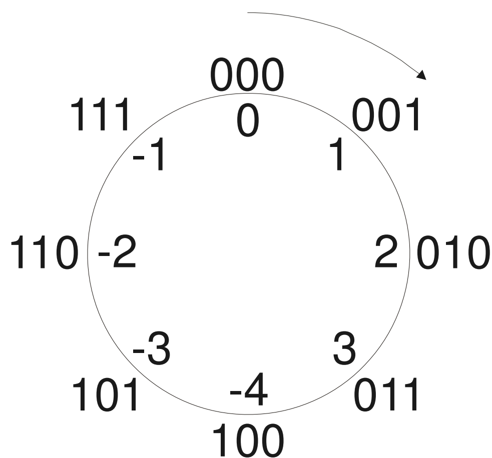


Рис 1.3 Круг целых чисел

При выполнении арифметических операций нужно контролировать выход за границы диапазона чисел (например: $-4+(-1)=100_{ДДК}+111_{ДДК}=011_{ДДК}=3$). Мы не будем разбирать, как это делается, т.к. эта тема больше подходит курсу ассемблера.

1.7. Шестнадцатеричная ССч

Помимо двоичной системы счисления часто используется шестнадцатеричная ССч. В ней числа записываются с помощью 16 цифр: 0,1,2,…,9, A, B, C, D, E, F. Цифрам A, B, C, D, E, F в десятичной ССч соответствуют числа 10, 11, 12, 13, 14, 15.

Для перевода числа в шестнадцатеричную ССч надо записать число n в виде $n=16^{k-1}\cdot d_{k-1}+16^{k-2}\cdot d_{k-2}+...+16^0\cdot d_0$, где все d_i - шестнадцатеричные цифры (0,1,….9, A, B, C, D, E, F). Теперь записывая подряд все цифры, получим запись заданного числа в 16-ричной ССч.

Например:

$132=16\cdot 8+16^0\cdot 4=84_{16}$

$78=16\cdot 4+16^0\cdot \text{E}=4E_{16}$

Шестнадцатеричные числа полезны, так как они позволяют упростить запись 2-ых чисел (см. упражнение № 17). Хорошо разобравшись в двоичной ССч, вы без труда сумеете выполнять арифметические операции и в 16-й ССч.

1.8. Более общая точка зрения на системы счисления

Когда мы переводили число n из десятичной ССч в двоичную, мы искали коэффициенты c_i в разложении $n=c_0 2^0+c_1 2^1+...+c_k 2^k+...$, а после этого упорядоченный

набор коэффициентов $(c_0, c_1, ..., c_k, ...)$ мы называли числом, соответствующим числу n в двоичной ССч (вообще-то числом мы называли $(..., c_k, ..., c_0)$[4], однако в этом параграфе удобнее поменять порядок следования коэффициентов на обратный).

Давайте теперь поставим более общую задачу: пусть задан упорядоченный набор $(a_0, a_1, ..., a_k, ...)$. Мы будем искать набор $(c_0, c_1, ..., c_k, ...)$ такой, что: $n = c_0 a_0 + c_1 a_1 + ... + c_k a_k + ...$

Сам набор $(c_0, c_1, ..., c_k, ...)$ будем называть разложением числа n по системе $(a_0, a_1, ..., a_k, ...)$.

Отмечу, что разложение числа может не существовать вообще, или оно может быть не единственно.

Например, найдем коэффициенты разложения числа 4 по системе $(2^0, 2^1, ..., 2^k, ...)$. Если c_i - натуральны, то таких разложений 3:

$4 = 4 \cdot 2^0$,

$4 = 2 \cdot 2^0 + 1 \cdot 2^1$

$4 = 0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2$

В новых обозначениях они выглядят так: $4 = (4, 0, ..., 0, ...) = (2, 1, 0, ..., 0, ...) = (0, 0, 1, 0, ..., 0, ...)$.

То, что разложение по системе $(2^0, 2^1, ..., 2^k, ...)$ для натуральных чисел существует всегда, очевидно. Для получения единственности в двоичной ССч ставится ограничение на величину коэффициентов: все они должны быть 0 или 1. Тогда выходит, что $4 = (0, 0, 1, 0, ..., 0, ...)$.

Поэтому введем такое определение ССч:

- Системой счисления над некоторым числовым множеством M назовем упорядоченный набор $A = (a_0, a_1, ..., a_k, ...)$ вместе с набором правил S, которые для любого числа $n \in M$ обеспечивают существование и единственность представления в виде суммы следующего вида: $n = c_0 a_0 + c_1 a_1 + ... + c_k a_k + ...$.
- Упорядоченный набор $(c_0, c_1, ..., c_k, ...)$ будем называть числом n, записанным в системе (A, S).

Обычные позиционные системы счисления – наиболее просты, поэтому неудивительно, что они появились самые первые. Менее тривиальным примером является фибоначчиева ССч.

1.9. Фибоначчиева ССч

- Последовательность Фибоначчи определяется так:

$f_0 = f_1 = 1$,

$f_{n+1} = f_n + f_{n-1}$, $n > 1$

Первые члены последовательности, соответственно, такие: 1, 1, 2, 3, 5, 8, 13…

- Числами Фибоначчи мы будем считать числа $f_1, f_2, f_3, ...$ (т.е. первую из единиц отбросим для удобства).

Названа эта последовательность в честь средневекового математика Фибоначчи, который пришел к ней, рассматривая задачу о размножении кроликов. Она состояла в следующем: по какому закону будет меняться численность кроликов, если: каждый месяц пара кроликов рождает новую пару и воспроизводить потомство кролики могут

[4] Наверное, причиной всему арабы, которые пишут справа налево.

начиная со 2-го месяца своего рождения. Можно проверить, что если изначально есть 1 пара только что родившихся кроликов, то количество пар кроликов в зависимости от месяца составит как раз последовательность Фибоначчи.

Чтобы вы освоились с этими числами, докажем простое свойство:

$$f_1 + f_2 + ... + f_n = f_{n+2} - 2$$

Доказательство:

Из определения ЧФ получаем:

$f_1 = f_2 - f_0$

$f_2 = f_3 - f_1$

...

$f_n = f_{n+1} - f_{n-1}$

Сложим числа $f_1, f_2, ..., f_n$. Получим:

$f_1 + f_2 + ... + f_n = (f_2 - f_0) + (f_3 - f_1) + (f_4 - f_2) + ... + (f_{n+1} - f_{n-1}) = -f_0 - f_1 + f_n + f_{n+1} = f_{n+2} - 2$

У ЧФ есть много других замечательных свойств. Некоторые из них приведены вам в качестве задач.

Теперь мы займемся тем, раде чего и затевали сыр-бор, т.е. фибоначчиевой ССч.

Лемма: $\forall n \in N$ **существует представление в виде суммы различных чисел Фибоначчи.**

Доказательство:

Пусть дано число k. Если k - ЧФ, то лемма доказана. В противном случае найдем максимальное из чисел Фибоначчи, которое меньше k. пусть это число - f_{i_0}.

Тогда $k = k_1 + f_{i_0}$. Теперь найдем максимальное число Фибоначчи, меньшее k_1 - пусть это - f_{i_1}. Тогда $k = f_{i_0} + f_{i_1} + k_2$. Замечу, что какое бы ни было число k, f_{i_0} и f_{i_1} не будут совпадать. Если бы это было бы не так, то тогда выполнялось бы неравенство $k > 2f_{i_0}$. Но $f_{i_0+1} < 2f_{i_0}$, а значит, f_{i_0} не является наибольшим числом Фибоначчи, меньшим k. Т.е. при любом k выполняется $f_{i_0} > f_{i_1}$.

Если на некотором s-м шаге мы получим число k_s, которое будет числом Фибоначчи, то мы доказали то, что хотели. А иначе и быть не может, т.к. для любого натурального числа r найдется число Фибоначчи, которое будет $\leq r$.

Итак, для любого числа k можно записать:

$$k = f_{i_0} + f_{i_1} + ... + f_{i_s} \quad (1)$$

Конец доказательства.

Мы с вами доказали, что любое число k можно представить в виде линейной комбинации чисел Фибоначчи: $k = c_1 f_1 + ... + c_n f_n$, где $c_i \in \{0,1\}, i = \overline{1,n}$. Значит, если мы выберем $A = (f_1, ..., f_n, ...)$ и укажем правило, по которому можно добиться однозначности разложения, то мы построим систему счисления.

Но что хорошо: доказывая предыдущую лемму мы использовали алгоритм, связанный с выделением на каждом шаге наибольшего фибоначчиевого слагаемого. Его мы и возьмем в качестве правила. Фибоначчиева ССч построена.

Давайте теперь напишем несколько представлений чисел в ФССч:

20=2+5+13=$f_2 + f_4 + f_6 = 0 \cdot f_1 + 1 \cdot f_2 + 0 \cdot f_3 + 1 \cdot f_4 + 0 \cdot f_5 + 1 \cdot f_6$.

Значит: $20_{10} = 010101_{ФССч}$
$39 = 5 + 34 \Rightarrow$ $39{=}00010001_{ФССч}$

1.10. Метод математической индукции

Часто приходится доказывать некоторые утверждения, касающиеся натуральных чисел. Например: показать, что $1^3 + 2^3 + ... + n^3 = (1 + 2 + ... + n)^2$ при всех $n \in N$.
Для доказательства можно использовать следующий метод:

1. База индукции: доказываем, что утверждение верно для $n = 1$.
2. Шаг индукции: в предположении, что утверждение верно для всех чисел $k = \overline{1,n}$ доказываем, что оно верно и для $k = n + 1$.

В результате из того, что утверждение верно при $n = 1$ следует, что оно верно и при $n = 2$, из того, что оно верно при $n = 1, 2$ следует, что оно верно и при $n = 3$ и т. д. Проделывая шаг индукции мы как бы доказываем сразу бесконечное количество утверждений.

Пример 1: Доказать, что $1^3 + 2^3 + ... + n^3 = \dfrac{n^2(n+1)^2}{4}$ **при всех** $n \in N$**.**

База индукции (БИ): при $n = 1$ выражение примет вид $1^3 = 1^2$
Предположение индукции (ПИ): предположим, что при всех $k = \overline{1,n}$ выполняется $1^3 + 2^3 + ... + k^3 = \dfrac{k^2(k+1)^2}{4}$.
Шаг индукции (ШИ): докажем, что равенство выполняется и при $k = n + 1$.

$$1^3 + 2^3 + ... + n^3 + (n+1)^3 = [\text{согласно} \quad ПИ] = \frac{n^2(n+1)^2}{4} + (n+1)^3 = \frac{(n+1)^2(n^2+4n+4)}{4} = \frac{(n+1)^2(n+2)^2}{4}$$

Пример 2: Доказать, что $n^3 + 5n \vdots 6$ **при** $n \geq 1$.
БИ: при $n = 1$ утверждение очевидно
ПИ: предположим, что для любого $k \leq n$ выполняется $k^3 + 5k \vdots 6$
ШИ: докажем, что утверждение верно и для $k = n + 1$:
$(n+1)^3 + 5(n+1) = n^3 + 3n^2 + 3n + 1 + 5n + 5 = (n^3 + 5n) + 3n(n+1) + 6$
Теперь:
$n^3 + 5n \vdots 6$ по ПИ,
$n(n+1) \vdots 2 \Rightarrow 3n(n+1) \vdots 6$
Следовательно: $(n^3 + 5n) + 3n(n+1) + 6 \vdots 6$. Утверждение доказано.

Пример 3: Доказать неравенство Бернулли: $(1-a)^n \geq 1 - na$**, где** $0 < a < 1$ **.**
БИ: при $n = 1$ неравенство очевидно
ПИ: предположим, что для любого $k \leq n$ выполняется $(1-a)^k \geq 1 - ka$
ШИ: докажем, что неравенство выполняется и для $k = n + 1$:
$(1-a)^{n+1} = (1-a)^n(1-a) \geq [\text{согласно ПИ и тому, что } a < 1] \geq (1-na)(1-a) = 1 - na - a + na^2 =$
$= 1 - (n+1)a + na^2 \geq 1 - (n+1)a$
Неравенство доказано.

Как видите, индукция – замечательный способ доказательства, однако чтобы применять его надо знать, что доказывать. Что бы мы делали, если бы не знали ответа заранее? К счастью, иногда утверждения можно довольно легко сформулировать, и тогда индукция всесильна. Кроме того, данный метод можно обобщить, доказывая утверждения не только для натуральных чисел, но и для более сложных множеств.

1.11. Логарифмы

Число x называется логарифмом числа a по основанию b, если $a = b^x$. Обозначается: $x = \log_b a$.

Например: $\log_2 8 = 3$, т.к. $8 = 2^3$, $\log_9 3 = \frac{1}{2}$, т.к. $3 = 9^{\frac{1}{2}}$.

Давайте рассмотрим несколько свойств логарифмов:

1. Сразу из определения вытекает, что $a = b^{\log_b a}$.
2. $\log_c a^b = b \log_c a$

Доказательство. Пусть $x = \log_c a^b \Rightarrow a^b = c^x \Rightarrow c^{\frac{x}{b}} = a \Rightarrow \frac{x}{b} = \log_c a \Rightarrow x = b \log_c a$.

3. $\log_c (ab) = \log_c a + \log_c b$

Доказательство. Пусть $x = \log_c a$, $y = \log_c b$.

Рассмотрим $c^{x+y} = c^x c^y = \left[\text{Из определения логарифма}\right] = ab$. Следовательно: $x + y = \log_c (ab)$.

1.12. Последовательности и прогрессии

- Последовательностью чисел называется упорядоченный набор чисел.
- Арифметической прогрессией называется последовательность чисел, в которой каждый следующий член отличается от предыдущего на постоянное число, именуемое разностью прогрессии, или $a_{n+1} = a_n + d = a_1 + nd$, $n > 1$

Например, натуральный ряд 1,2,3, … , n, ... – арифметическая прогрессия, начинающаяся с 1, и с разностью прогрессии =1.

Давайте выведем формулу для суммы первых n членов арифметической прогрессии. Можно это сделать, замечая, что сумма первого и последнего слагаемого равна сумме второго и предпоследнего слагаемого и т.д. Но это доказательство слишком известно, поэтому мы пойдем другим путем. Сначала преобразуем сумму:

$a_1 + a_2 + ... + a_n = a_1 + a_1 + d + ... + a_1 + (n-1)d = na_1 + d(1 + 2 + ... + (n-1))$

Обозначим: $S_1(n-1) = 1 + 2 + ... + (n-1)$. Теперь вычислим $S_1(n-1)$.

Сделать это можно таким красивым способом: чтобы вычислить $S_1(n)$ будем отталкиваться от $S_2(n) = 1^2 + 2^2 + ... + n^2$

$S_2(n) = 1^2 + (1+1)^2 + ... + (n-1+1)^2 = 1^2 + (1^2 + 2 \cdot 1 + 1^2) + (2^2 + 2 \cdot 2 + 1^2) + ... + ((n-1)^2 + 2(n-1) + 1^2) =$

$= 1^2 + 2^2 + ... + (n-1)^2 + n + 2(1 + 2 + ... + (n-1))$.

А теперь ясно, что $n + 2(1 + 2 + ... + (n-1)) = n^2$ и

$$S_1(n-1) = 1 + 2 + ... + (n-1) = \frac{n(n-1)}{2}$$

Этот способ выглядит сложнее, чем тот, о котором я говорил вначале, но с его помощью можно получить рекуррентные формулы для $S_p(n) = 1^p + 2^p + ... + n^p$.

Теперь мы можем записать формулу суммы первых n членов арифметической прогрессии:

$$a_1 + a_2 + \ldots + a_n = na_1 + d\frac{n(n-1)}{2} = \frac{a_1 + (n-1)d}{2}n = \frac{a_1 + a_n}{2}n$$

- Геометрической прогрессией называется последовательность чисел, в которой каждый следующий член отличается от предыдущего в постоянное число раз, именуемое знаменателем прогрессии, или $b_{n+1} = qb_n$, $n > 1$

Теперь давайте выведем формулу для суммы первых n членов геометрической прогрессии.

$$b_1 + b_2 + \ldots + b_n = b_1 + qb_1 + q^2 b_1 + \ldots + q^{n-1} b_1 = b_1(1 + q + q^2 + \ldots + q^{n-1})$$

А теперь воспользуемся формулой

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

(Ее можно проверить, просто раскрывая скобки и приводя подобные слагаемые).
В результате получим искомую формулу:

$$b_1 + b_2 + \ldots + b_n = \frac{b_1(1-q^n)}{1-q}$$

Если $|q| < 1$, то можно вычислить и сумму бесконечного числа членов прогрессии. Так как $|q| < 1$, то $|q|^n \xrightarrow[n\to\infty]{} 0$, то получим формулу:

$$b_1 + b_2 + \ldots + b_n + \ldots = \frac{b_1}{1-q}$$

Задачи[5]

1. Какое наибольшее отрицательное рациональное число?
2. Сложите числа $0.(5)$ и $0.(6)$ без использования обыкновенных дробей.
3. Переведите числа 1094 и 1024 из десятичной в двоичную систему.
4. Переведите шестнадцатеричные числа BAD, FEED в двоичную систему.
5. Выведите формулу для вычисления количества цифр натурального числа, записанного в 10-й ССч.
6. Обобщите результат задачи 5 на случай произвольной системы счисления.
7. Докажите, что число 0,123456789101112… - иррационально.
8. Найдите ошибку в следующем доказательстве: «Теорема: пусть $a > 0$. Тогда для любого числа $n > 0$, $n \in N$ выполняется равенство $a^{n-1} = 1$. Доказательство: БИ: $n = 1$ $\Rightarrow$ $a^0 = 1$. Пусть теперь, по предположению индукции, теорема верна для $k = \overline{1,n}$. Докажем, что она выполняется и для $k = n+1$.
   $a^{(n+1)-1} = a^n = \dfrac{a^{n-1} \cdot a^{n-1}}{a^{(n-1)-1}} = [\text{согласно ПИ}] = \dfrac{1 \cdot 1}{1} = 1$.»
9. Следующее доказательство по индукции выглядит корректным, но по непонятной причине для $n = 6$ левая часть уравнения дает $\frac{1}{2} + \frac{1}{6} + \frac{1}{12} + \frac{1}{20} + \frac{1}{30} = \frac{5}{6}$, а правая $\frac{3}{2} - \frac{1}{6} = \frac{4}{3}$. В чем же дело?

[5] Задачи 8, 9 взяты из книги Дональда Кнута «Искусство программирования» (том 1), задача 18 – из книги Дьердя Пойа «Математическое открытие».

«Теорема: $\frac{1}{1\cdot 2}+\frac{1}{2\cdot 3}+...+\frac{1}{(n-1)\cdot n}=\frac{3}{2}-\frac{1}{n}$

Доказательство: БИ: при $n=1$ $\frac{3}{2}-\frac{1}{1}=\frac{1}{1\cdot 2}$. Предполагая, что теорема верна для всех $k\le n$, получим:

$\frac{1}{1\cdot 2}+\frac{1}{2\cdot 3}+...+\frac{1}{(n-1)\cdot n}+\frac{1}{n\cdot(n+1)}=\frac{3}{2}-\frac{1}{n}+\frac{1}{n\cdot(n+1)}=\frac{3}{2}-\frac{1}{n+1}$. Теорема доказана»

10. Докажите обобщение неравенства Бернулли:
    $(1-a_1)(1-a_2)...(1-a_n)\ge 1-(a_1+a_2+...+a_n)$, где $0<a_i<1$, $i=\overline{1,n}$
11. Докажите по индукции: для $n\ge 10$ выполняется $2^n>n^3$.
12. Докажите: $7^n+12n+17\,\vdots\,18$ при любом натуральном n.
13. Докажите: $\left(a_1+a_2+...+a_n\right)^2=a_1^2+a_2^2+...+a_n^2+2a_1a_2+2a_1a_3+...+2a_{n-1}a_n$
14. Докажите, что число $\log_3 15$ - иррационально.
15. Пусть задана периодическая дробь в k-ичной системе счисления в виде $f_k=0.(s_k)$, где s_k - некоторое число, заданное в k-ичной ССч. Предложите простую формулу перевода числа f_k в 10-ную ССч.
16. Возьмите два числа в двоичной системе и разделите их столбиком
17. Напишите простой алгоритм перевода 2-ых чисел в 16-ричное представление и обратно.
18. **(!)** Если вы уже знаете закон, то доказать его по индукции – дело техники. Но зачастую вы заранее не знаете зависимости – вы должны угадать ее! Если закон достаточно сложен, то угадать его непросто. Вот один из примеров:
    $y=x^{-1}\ln x$. Найти $y^{(n)}$. Давайте поэкспериментируем:
    $y'=-x^{-2}\ln x+x^{-2}$
    $y''=2x^{-3}\ln x-3x^{-3}$
    $y'''=-6x^{-4}\ln x+11x^{-4}$

Теперь давайте попробуем угадать сам закон. Общий вид первого слагаемого определить легко, а вот коэффициент при втором слагаемом сразу в голову не приходит. Можно записать так:

$y^{(n)}=(-1)^n n!x^{-n-1}\ln x+(-1)^{n-1}c_n x^{-n-1}$, $n!=1\cdot 2\cdot...\cdot n$, $n>0$.

Докажите это соотношение, а затем найдите коэффициент c_n.

19. **(!)** Пусть $\{f_n\}$ - последовательность Фибоначчи. Найдите $\varphi=\lim\limits_{n\to\infty}\frac{f_{n+1}}{f_n}$
    - Сначала найдите число φ в предположении, что предел существует.
    - Докажите, что число φ действительно является пределом $\frac{f_{n+1}}{f_n}$.
20. **(!)** Докажите, что полученное число φ из упражнения 19 представимо в виде $\varphi=1+\cfrac{1}{1+\cfrac{1}{1+...}}$
21. **(!)** С помощью метода, который был применен для вычисления суммы членов арифметической прогрессии, выведите рекуррентную формулу для $S_p(n)=1^p+2^p+...+n^p$

Глава 2: Первые шаги

2.1. Базовые концепции процедурного программирования

Введем некоторые основные понятия процедурного программирования:

- **Алфавит** – множество символов, допустимых в языке.
- **Слово** – набор символов, отделенный с обеих сторон пробелами.
- **Переменная** – именованная область памяти, значение которой может изменяться во время работы программы.
- **Константа** – именованная область памяти, значение которой не может изменяться во время работы программы.
- **Идентификатор** – слово, которое может быть использовано в качестве названия для переменной, константы, типа данных и других объектов программного кода.
- **Тип данных** – идентификатор, определяющий объем памяти, который занимает переменная, а также интерпретацию данных, записанных в переменной и операции, которые можно совершать над переменными данного типа.

Все эти понятия отсутствуют в машинном языке. Для процессора нет никаких констант, переменных и тем более типов. Есть лишь устройства, например, память, и команды, которые можно этим устройствам посылать – и ничего больше.

В языках программирования существует некоторый стандартный набор типов данных, с которыми можно работать (напр. целые и вещественные числа, символы). Кроме того, есть возможности строить на основе этих простых типов более сложные, которые состоят из набора переменных разных типов.

Операции

Прежде, чем давать определение операции в программировании, рассмотрим, как операция определяется в математике.

- **Бинарной операцией** над множеством X называется функция, которая двум элементам множества X ставит в соответствие некоторый (единственный) элемент множества X.

Например, бинарной операцией является операция сложения на множестве целых чисел, т.к. сумма двух целых чисел – тоже целое число. А операция вычитания на множестве натуральных чисел не будет являться бинарной операцией, т.к. разностью двух натуральных чисел не всегда является натуральное число.

Естественно, кроме бинарных операций можно рассматривать **унарные** (т.е. операции, которые одному элемента множества ставят в соответствие единственных элемент того же множества), **тернарные** (операции, которые трем элементам множества ставят в соответствие единственный элемент этого множества, и.т. д.

В программировании бинарную операцию обычно определяют иначе:

- **Бинарная операция** – это функция, которая двум переменным, совместимым по типу, ставит в соответствие значение некоторого типа (обычно тип результата совпадает с типом одной из переменных-операндов).

Совместимость по типу может определяться для каждого языка по-разному. Набор формальных правил, задающих совместимость по типу для TP, мы изучим позднее.

Операторы

- Оператор – языковая конструкция, предназначенная для описания действий алгоритма.

Операторы делятся на простые и составные.

- Простой оператор – оператор, не содержащий в себе никаких других операторов.
- Составной оператор – оператор, не являющийся простым.

Самым важным из простых операторов является оператор присваивания, который записывает в переменную результат некоторого выражения или значение другой переменной. В выражении могут использоваться различные операции над переменными и константами.

Из структурных операторов наиболее важны условные операторы и операторы цикла. Условные операторы позволяют проверять различные условия, и в зависимости от них выполнять те или иные действия. Операторы цикла позволяют совершать повторяющиеся действия.

Структура программного кода

Прежде, чем использовать переменные, константы или типы данных, надо сначала объявить их. В некоторых языках (таких, как C, Java) объявления переменных и их использование не разделяются особо. В других (как TP, Delphi) есть специальная часть программы, в которой находятся все объявления переменных, типов, подпрограмм, которые будут использоваться в дальнейшем.

Исполняемая часть программы состоит из набора операторов, которые надо выполнить.

- Языки, в которых программа представляет собой набор операторов, называются **императивными языками**.

Императивными языками являются все процедурные языки, а также машинные языки и ассемблеры.

В процедурных языках программа разбивается на фрагменты, которые называются подпрограммами. Для того, чтобы вызвать подпрограмму используется оператор вызова. Его синтаксис в языках неодинаков. Например, в языке Fortran (от англ. Formulae Translator) чтобы вызывать подпрограмму UnterPr, надо написать следующую строку:

```
CALL UnterPr
```

В TP подпрограмма вызывается просто по имени:

```
UnterPr;
```

В подпрограммы можно передавать параметры (например, в подпрограмму вычисления синуса числа должно передаваться само число, синус которого надо вычислить).

В TP параметры, которые передаются в подпрограмму, записываются в скобках после называния подпрограммы. Сами же подпрограммы в TP делятся на 2 класса:

- процедуры - подпрограммы, которые не возвращают значений.
- функции – подпрограммы, возвращающие значение.

Теперь, немного подкрепившись теорией, можно приступать к практическим занятиям. На протяжении всей первой части книги мы будем изучать процедурное программирование на TP, параллельно рассматривая многие задачи и алгоритмы, которые необходимо знать программисту.

2.2. Что нам понадобится для работы

Для того чтобы писать программы, нам нужен текстовый редактор для написания кода и компилятор, с помощью которого мы сможем переводить этот код в машинные инструкции. Как правило, создаются системы разработки приложений, которые объединяют в себе обе эти функции. Кроме того, если возникают ошибки при компиляции программы, то среда сообщает вам об этом, указывает место ошибки и ее предположительную причину.

Для того, чтобы писать программы на Паскале, вам понадобится один из следующих компиляторов: TP 7.0, TP 7.1, FreePascal, Borland Pascal (BP). Особых отличий между ними нет, поэтому хотя я буду называть язык программирования TP, но все то же самое, если нет особых комментариев, походит и для других версий языка. В TP 7.1 (или более ранних версиях) запускающий файл называется turbo.exe, во FreePascal – fp.exe, в BP – bp.exe. Запустите этот файл, и пред вашими очами появится синий экран. Это – текстовое поле, в котором мы и будем писать наши программы.

В этой главе мы изучим основы работы в среде TP и напишем несколько простых программ. Основные комбинации клавиш для работы в среде TP такие:

- F9 – компиляция программы
- Ctrl+F9 – компиляция и запуск программы
- F2 – сохранить файл
- F3 – открыть файл
- Alt+F5 – просмотр результатов работы. В Windows XP эта комбинация клавиш может не сработать, поэтому вам придется зайти в меню Debug и выбрать опцию User Screen.

По умолчанию программы сохраняются в том каталоге, где находится запускающий файл TP.

2.3. Первые программы

Нашей первой программой будет вывод простого сообщения на экран.

Пример 1: Вывод сообщения на экран.

```
BEGIN
  writeln('Начало пути');
END.
```

Исполняемая часть программы начинается со слова BEGIN.

Во второй строке для вывода сообщения ‘Начало пути’ была использована процедура writeln. Writeln печатает подряд все параметры, которые ей передаются (в примере параметр один – строка ‘Начало пути’) и затем переводит курсор на следующую строку.

Словом END. заканчивается исполняемая часть программы. Ставить точку надо обязательно.

Замечание 1: не забывайте брать выводимое сообщение в одинарные кавычки, иначе компилятор выдаст сообщение об ошибке, например:
(Error 8: String constant exceeds line), если вы поставите лишь открывающую кавычку, или
(Error 5: Syntax error), если вы поставили вместо одинарных кавычек двойные.

Не стоит запоминать все сообщения об ошибках на память; я их привел лишь для того, чтобы вы не пугались их при написании первых программ.

Заметьте, что в примере во 2-й строке сделан отступ в два пробела. Хотя это делать не обязательно, но использование отступов увеличивает наглядность программы.

В результате работы программы на экране появится сообщение «Начало пути», а курсор переместится на следующую строку.

Теперь мы напишем программу, которая будет вычислять значение простого математического выражения, но сначала рассмотрим, как записываются знаки арифметических операций в ТР.

Арифметическая операция	Запись операции на ТР	Алгебраическое выражение	Выражение на ТР
Сложение	+	a+c	a+c
Вычитание	-	b-c	b-c
Умножение	*	Vf	V*f
Деление	/	$\frac{d}{f}$	d/f

Примеры записи алгебраических выражений в ТР:

В математике	В ТР
$\frac{a+b-cd}{f+gds}$	(a+b-c*d)/(f+g*d*s)
$a+b-\frac{cd}{f+gds}$	a+b-c*d/(f+g*d*s)

Замечание: помните о старшинстве операций: операции деления и умножения имеют более высокий приоритет, чем операции сложения и вычитания, поэтому не забывайте ставить скобки, если они нужны.

В дальнейшем для удобства работы с приведенным в книге программным кодом, строки будут нумероваться. Но помните, что нумерация сделана лишь для удобства, а при написании кода нумеровать строки нельзя.

Поэтому вместо

```
4 :   a=5;
5 :   b=2;
```

вы должны писать в TP так:

```
a=5;
b=2;
```

Пример 2: Вычисление S=(a+b)*m.

```
1 : program Progr1_2;  {S=(a+b)*m;}
2 : const {В разделе const мы с вами будем объявлять константы}
3 :   m=2; {Объявляем константу m=2}
4 :   a=5;
5 :   b=2;
6 : var {В разделе var (сокращение от variable - переменная)
7 :     мы будем объявлять переменные}
8 :   s:integer;{Объявление целочисленной переменной s}
9 : BEGIN
10:   S:=(a+b)*m; {Присваиваем переменной s значение (a+b)*m }
11:   write(S); {Печатаем на экран значение переменной s}
12: END.
```

Эта программа по своей структуре уже сложнее предыдущей. В ней кроме исполняемой части (которая начинается со слова Begin, и заканчивается словом end.), есть раздел описаний, который находится перед исполняемой частью и состоит из разделов program, const, var.

- Раздел program начинается с ключевого слова program, за которым через пробел следует название программы (которое вы придумываете сами). Обычно программисты этот раздел пропускают, т.к. зачастую предназначение программы не опишешь одним словом, и все равно приходится писать комментарий (о том, что это такое, мы поговорим ниже).
- Выражения в фигурных скобках – это комментарии (например {S=(a+b)*m;}). Комментарий – это текст, который игнорируется компилятором и предназначенный исключительно как пояснения к программному коду, т.е. если вы их поставите или, наоборот, уберете, то программа при этом никак не изменится. Применение комментариев поможет вам быстрее разбираться в ваших программах. Если вы пишете одну часть программы сегодня, а другую через месяц, то, приступая ко второй части программы, вы можете совершенно забыть о том, что было написано в первой части. Кроме того, хорошо прокомментированную программу легче отлаживать (т.е. искать ошибки), что экономит вам время и силы.
- В разделе const (от англ. constant - постоянная) описываются константы. Чтобы описать константу, вы должны написать ее имя, затем знак равенства и значение константы. В конце описания должна стоять ‘;’. В примере 2 были описаны 3 целочисленные константы m,a,b со значениями 2, 5, 2 соответственно.
- За разделом const следует раздел var (от англ. variable – переменная). В этом разделе объявляются переменные. Описываются переменные следующим образом: сначала название переменной, затем знак ‘:’, затем тип переменной. В примере 2 была описана одна переменная типа integer (целого типа). Подробнее о типах данных мы поговорим в следующих главах.
- Предыдущие 3 раздела называются разделами описаний. За ними следует исполняемая часть программы (которая, как мы уже знаем, начинается с ключевого слова BEGIN и заканчивается словом END.)

- В строке 10 стоит оператор S:=(a+b)*m;. ':=' - оператор присваивания. Он записывает в переменную, стоящую в левой части оператора значение, стоящее в его правой части. В нашем случае сначала будет вычислено значение выражения (a+b)*m, а затем это значение будет присвоено переменной S.
- В строке 11 значение переменной S выводится на экран процедурой write. Эта процедура отличается от writeln лишь тем, что после вывода сообщения курсор не переводится на следующую строку.

Замечание 1: Заметьте, что в строке 11 переменная S не взята в кавычки. Если бы вместо write(S) было написано write('S'), то мы бы увидели на экране не значение переменной S, а просто букву S (Проверьте!).

Замечание 2: вы видите, что все названия переменных, констант и т.п. написаны с помощью латинского алфавита. Дело в том, что TP не понимает кириллицу, поэтому если вы знаете какой-либо язык, использующий латинский алфавит, то пишите на нем, а если нет, то выучите немецкий и дело с концом. Чтобы помочь вам в этом, в дальнейшем я буду использовать в качестве идентификаторов немецкие названия (естественно, в комментариях я буду давать вам их перевод).

Записывать алгебраические выражения мы научились и теперь можем двигаться дальше. Нашей следующей целью будет использование встроенных математических функций. С большинством из них вы знакомы (или еще познакомитесь). Мы же рассмотрим функции целой и дробной части, которые стоят особняком, т.к. они не являются непрерывными функциями.

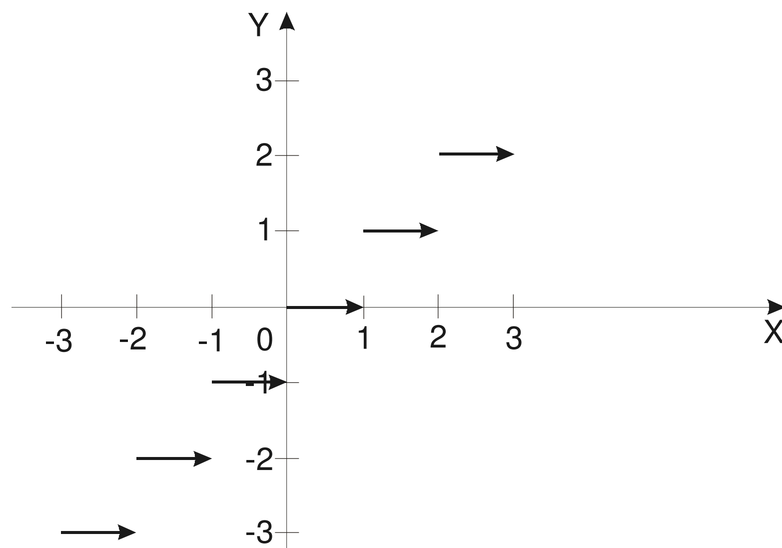


Рис 2.1. График функции $y=[x]$

- Целой частью вещественного числа x (обозначается $[x]$) называется наименьшее число из двух ближайших к x целых чисел. Целой частью целого числа является само число.

Пример: [10.13]=10, [3.99]=3, [-1.13]=-2, [-20]=-20.

На рис. 2.1 изображен график функции $y=[x]$.

Стрелки означают, что функция стремится к некоторому значению аргумента, но не достигает его, например: на полуинтервале [0;1) значение функции равно 0, а в точке

x=1 уже единице, поэтому стрелка, ведущая из точки (0,0) в (1,0) самой точки (1,0) не достигает.

Иногда вместо $\longrightarrow$ используют обозначение $—\!\circ$ и точки, которые обведены в кружок, называют выколотыми.

- Дробная часть числа x (обозначается $\{x\}$) определяется так: $\{x\} = x - [x]$.

{0.234}=0.234, {5.23}=0.23, {5}=0, {-3.25}=0.75.

В зарубежной литературе нет понятия целой части. Вместо этого рассматриваются следующие функции:

- $y = \lfloor x \rfloor$ - то же, что и целая часть. Называется она “пол”.
- $y = \lceil x \rceil$ - наименьшее целое число, большее числа x. Называется она «потолок».

Запись математических функций в ТР

В математике	В ТР	Название функции
|x|	Abs(x)	модуль
x^2	Sqr(x)	квадрат числа
$\sqrt{x}$	Sqrt(x)	квадратный корень числа
$\cos x$	Cos(x)	косинус
$\sin x$	Sin(x)	синус
$arctg(x)$	Arctan(x)[6]	арктангенс
$\ln x$	Ln(x)	натуральный логарифм
e^x	Exp(x)	экспонента
$\{x\}$	Frac(x)	дробная часть
$[x]$	Int(x)	целая часть
π	Pi	число π

Замечу, что функции на самом деле функции Int(x) и Frac(x) работают так:

$$\operatorname{int}(x) = \begin{cases} \lfloor x \rfloor, x > 0 \\ \lceil x \rceil, x \le 0 \end{cases} \qquad frac(x) = \begin{cases} \{x\}, x > 0 \\ \{x\} - 1, x \le 0, x \notin Z \\ 0, x \in Z \end{cases}$$

В ТР реализованы не все стандартные функции, поэтому ниже приведены выражения других элементарных функций через функции, реализованные в ТР.

$$a^x = e^{x \ln a}$$

$$shx = \frac{e^x - e^{-x}}{2}$$

(гиперболический синус)

$$chx = \frac{e^x + e^{-x}}{2}$$

$$\arcsin x = arctg \frac{x}{\sqrt{1-x^2}} \text{ (арксинус)}$$

$$\arccos x = arctg \frac{\sqrt{1-x^2}}{x} \text{ (арккосинус)}$$

$$arcctgx = arctg \frac{1}{x} \text{ (арктангенс)}$$

[6] В англоязычных странах тангенс обозначается не tg, а $\tan$, а котангенс – не ctg, а $\cot$.

(гиперболический косинус)

$\log_a b = \frac{\ln b}{\ln a}$

Теперь, научившись работать с константами и переменными, и с легкостью орудуя формулами, мы можем заняться моделированием равномерного прямолинейного движения автомобиля.

Напомню, на всякий случай, формулы вычисления скорости и пройденного пути при прямолинейном равноускоренном движении:

$v(t) = v_0 + at$

$S(t) = S_0 + v_0 t + \frac{at^2}{2}$, где

v_0 - скорость автомобиля в момент времени $t = 0$

S_0 - координата автомобиля в момент времени $t = 0$

a - ускорение автомобиля

$v(t), S(t)$ - скорость и координата тела в момент времени t.

В нашей задаче S_0=0.

Пример 3: Вычисление конечной скорости и пройденного автомобилем пути, если заданы ускорение, начальная скорость и время движения автомобиля.

```
1 : uses CRT; {Модуль, в котором находится процедура очистки экрана}
2 : const
3 :   a=1; {ускорение}
4 :   t=60; {время движения}
5 :   v0=20;{начальная скорость машины}
6 : var
7 :   S:real;{объявление вещественной переменной S}
8 :   v:real;{объявление вещественной переменной v}
9 : BEGIN
10:   Clrscr; {процедура, очищающая экран}
11:   v:=v0+a*t;{находим скорость в конце пути}
12:   S:=v0*t+(a*sqr(t)/2); {находим путь, пройденный автомобилем}
13:   writeln('Скорость в конце пути:',v,' м/с');
14:   writeln('За время движения машина пройдет ',S,' м');
15:   readln;{ждем, пока пользователь ждет нажатия Enter}
16: END.
```

В этой программе добавился еще один раздел описаний: uses. В нем описываются модули (коллекции подпрограмм), которые будут использоваться вами в программе. В данном примере мы подключили модуль CRT (эта аббревиатура расшифровывается как Cathod Radio-Tube, в переводе с английского - электронно-лучевая трубка, ЭЛТ) для того, чтобы мы могли использовать процедуру очистки экрана ClrScr (сокращение от Clear Screen – очистить экран) (см. строку 10).

В разделе var были объявлены 2 вещественные переменные (тип real). Дело в том, что при вычислении пройденного пути мы будем делить число на 2, и результатом будет вещественное число, поэтому целые типы данных использовать нельзя (иначе возникнет ошибка несоответствия типов (Error 26: Type mismatch)).

В строке 12 программа вычисляет длину пути, пройденного машиной за время t.

Печатаются данные с помощью уже знакомой нам процедуры writeln, однако на этот раз параметров мы ей передали не один, а несколько. В качестве параметров можно передавать как константы, так и переменные. Если в процедуру writeln передать константу, то она ее и напечатает, а если передать переменную, то будет напечатано значение переменной. Перевод строки будет сделан лишь после того, как будут выведены все аргументы функции.

Например, строка 13 данного примера выведет на экран строку "Скорость в конце пути:", затем значение переменной v, потом строку "м/с", и потом переведет курсор на следующую строку.

Процедура readln в строке 15 останавливает работу программы и ждет, пока пользователь нажмет клавишу Enter.

Переменные v и S – вещественные, и поэтому они печатаются в особом формате. Например, 3.23234E+4 означает $3.23234 \cdot 10^4$. В общем случае $cEd = c \cdot 10^d$.

Замечание: Не забывайте подключать модуль Crt, если вы используете в вашей программе подпрограммы из этого модуля.

2.4. Ввод значений переменных с клавиатуры. Процедура Readln(x)

Пришло время модернизировать наши программы так, чтобы мы могли вводить входные данные с клавиатуры, а не делать их константами. Ввод значений осуществляется с помощью процедур read(x) и readln(x), где x – имя переменной, значение которой мы должны ввести. Различия между этими процедурами мы обсудим позже, когда будем работать с массивами. А пока что будем использовать процедуру readln(x).

Процедура readln(x) дает возможность пользователю ввести с клавиатуры значение переменной x. Вы можете вводить число, стирать его клавишей Backspace, снова вводить и т.д. Окончание ввода числа – нажатие клавиши Enter. После этого введенное значение записывается в переменную, которая находится в скобках.

Замечание 1: Если вы вместо числа введете строку, которая числом не является, то будет выдана ошибка (Error 106: Invalid Numeric Format).

Замечание 2: В ТР десятичная точка – это символ '.', а не ','. Поэтому если вы введете значение 5,5 – то компилятор вам выдаст ошибку, т.к. для ТР 5,5 – это не число. Надо писать 5.5.

Замечание 3: Если вы вместо целого числа введете вещественное, то будет выдана ошибка (Error 106: Invalid Numeric Format).

Пример 4: Программа, вычисляющая по сумме вклада и процентной ставке банка проценты по вкладу в конце месяца.

```
1 : uses Crt;
2 : var
3 :   Einlage:real; {Вклад}
4 :   ProzentSatz:real;{Процентная ставка}
5 :   Einkommen:real; {Доход}
6 : BEGIN
7 :   write('Введите размер вклада (в грн): ');
8 :   readln(Einlage); {вводим размер вклада}
9 :   write('Введите процентную ставку банка (в % годовых): ');
10:   readln(ProzentSatz);{Вводим процентную ставку банка}
```

```
11:   Einkommen:=Einlage*ProzentSatz/(12*100);
12:   write('Ваш доход в конце месяца: ',Einkommen,' грн.');
13:   readln;
14: END.
```

В строке 7 мы выводим пользователю сообщение о том, какую информацию он должен ввести. Строка 8 требует от нас ввести значение переменной Einlage, а строка 10 – значение переменной ProzentSatz.

2.5. Несколько заключительных слов

Вы знаете, что в результате работы компилятора программный код на языке высокого уровня (ЯВУ) преобразуется в программу - набор команд на машинном языке. Все константы и переменные, которые мы объявляем в программе, тоже являются ее неотъемлемой частью, поэтому компилятор на этапе компиляции программы выделяет под каждую переменную и константу определенное место в памяти (после этого каждой переменной и константе соответствует место в памяти, где хранится ее значение) и инициализирует константы начальными значениями. Оператор присваивания (a:=b) будет преобразован компилятором в последовательность команд ЦП, которые в область памяти, в которой хранится значение переменной a, запишут данные, хранящиеся в области памяти, в которой хранится значение переменной b. Т.е. для компилятора названия переменных – это ссылки на место в памяти, где хранится значение переменной.

Если в операторе a:=b a – константа, то на этапе компиляции будет выдана ошибка, т.к. именно компилятор отслеживает попытки изменения константных данных.

Все встроенные подпрограммы, которые мы использовали в этой главе (включая математические функции) содержатся в модуле System. Этот модуль подключается к каждой программе автоматически. Все остальные модули вы должны подключать самостоятельно.

При нажатии комбинации клавиш Ctrl+F9 компилятор генерирует программу, которую он запускает сразу на выполнение. При этом не создается exe-файл, в котором хранится написанная программа. Если вы хотите, чтобы TP не только запускал программу на выполнение, но и создавал exe-файл, надо зайти в раздел Compile->Destination и вместо Memory установить значение Disk.

Задачи

1. Напишите программу, которая выводит на экран ваши имя и фамилию, место учебы (работы), ВУЗ, в который собираетесь поступать. (Если хотите, можете продолжить список).
2. Вы – главный менеджер украинского филиала компании DedMoroz&Co. Наступила зима, и главный украинский Дед Мороз должен готовиться к раздаче подарков. Ему нужно знать, сколько сладостей нужно заготовить в этом году. Вы по долгу службы обязаны предоставить шефу все необходимые данные: количество подарков, суммарный вес подарков, стоимость одного подарка, и суммарную стоимость всех подарков. Все исходные данные, а именно: масса одного подарка, средняя стоимость килограмма сладостей, количество детишек в стране вводятся с клавиатуры. Только

не печатайте на экране одни цифры. Босс должен увидеть хороший отчет, а не то он может разозлиться на вас и даже выгнать с работы…

3. По введенным двум катетам вычислите гипотенузу, площадь и периметр прямоугольного треугольника, и выведите их на экран.
4. По введенному радиусу программа должна выводить на экран длину окружности, площадь круга, объем шара.

Длина окружности	Площадь круга	Объем шара
$L = 2\pi R$	$S = \pi R^2$	$V = \frac{4}{3}\pi R^3$

Во всех формулах R – радиус, а число π - это иррациональное число, и оно $\approx 3{,}1415$

Значение числа π можно получить с помощью встроенной функции Pi.

Например, чтобы в переменную L записать длину окружности радиуса R, то можно записать так:

L:=2*Pi*R;

5. Вам заданы радиус основания цилиндрической железной бочки, ее высота и толщина (все в метрах). Вы должны вычислить, какой объем бочки, и какова масса пустой бочки (в кг). Плотность железа – 7800 $\frac{кг}{м^3}$.

Объем цилиндра вычисляется по формуле $V = \pi R^2 H$, где R – радиус цилиндра, H – его высота.

6. Начертите график функции $y = \{x\}$.
   - Функция $y = f(x)$ называется периодической, если существует такое число $T > 0$, что для любого x из области определения функции $f(x)$ выполняется условие $f(x+T) = f(x)$.
   - Минимальное из чисел T называется периодом функции $y = f(x)$.
7. Докажите, что функция $y = \{x\}$ – периодическая, и ее период =1.
8. Верно ли, что $\{-x\} = 1 - \{x\}$. Объясните.
9. Докажите неравенство $\{x + y\} \le \{x\} + \{y\}$.
10. Исходя из результата предыдущей задачи, докажите, что $[x + y] \ge [x] + [y]$.
    - Функция $y = f(x)$ называется аддитивной, если $f(x+t) = f(x) + f(t)$.
11. Обобщите результаты задач 4,5, доказав следующее утверждение:

Если аддитивная функция $f(x)$ представима в виде $f(x) = f_1(x) + f_2(x)$, то из того, что $f_1(x + y) \le f_1(x) + f_1(y)$ следует $f_2(x + y) \ge f_2(x) + f_2(y)$, и наоборот.

12. Пусть $d(x)$ - расстояние между числом x и ближайшим целым числом. Для любого натурального n вычислите: $F_n = \sum_{m=1}^{6n-1} \min\left(d\left(\frac{m}{6n}\right),\ d\left(\frac{m}{3n}\right)\right)$.

Глава 3: Использование оператора if

3.1. Немного справочной информации

В начале предыдущей главы мы ввели понятия алфавита, идентификатора, переменной, константы. Сейчас мы рассмотрим, как эти понятия реализованы в ТР.

Алфавит

Алфавит языка Турбо Паскаль включает буквы, цифры, специальные символы, пробелы.

Буквы — это буквы латинского алфавита от *a* до *z* и от *A* до Z, а также знак подчеркивания _.

Специальные символы Турбо Паскаля:

+ - * / = , ' . : ; < > [ ] ( ) { } ^ @ $ #

Из специальных символов состоят также некоторые пары символов, которые обладают собственным смыслом:

<> <= >= := (* *) (. .)

Цифры — десятичные цифры от 0 до 9.

Пробелы в ТР используются в качестве разграничителей.

То, что алфавит включает в себя лишь указанные выше символы, не означает, что вы можете использовать в программном коде только их. Вы знаете, что в строковых константах можно использовать и кириллицу. Однако строковые константы не являются частью языка – это просто данные, которыми оперирует программа.

Идентификаторы в ТР могут состоять из букв или цифр, причем первым символом должна обязательно быть буква.

В ТР между строчными и заглавными буквами различий нет, поэтому, например, следующие идентификаторы будут считаться одинаковыми:

Herz herz

F2e3 f2E3

Некоторые слова, которые называются **зарезервированными**, используются в языке для обозначения операторов, объявлений и т.д. Они не могут использоваться в качестве идентификаторов.

В таблице приведен их полный список.

and	end	nil	shr
asm	file	not	string
array	for	object	then
begin	function	of	to
case	goto	or	type
const	if	packed	unit
constructor	implementation	procedure	until
destructor	in	program	uses
div	inline	record	var
do	interface	repeat	while
downto	label	set	with
else	mod	shl	xor

Для того, чтобы задать специальные свойства подпрограммам, используются **стандартные директивы.** Ниже приведен список стандартных директив:

absolute far near
assembler forward private
external interrupt virtual

По умолчанию зарезервированные слова и стандартные директивы в окне редактора TP выделяются белым цветом (но в TP можно менять цвета фона и символов, поэтому вполне возможно, что в вашем TP кто-то уже полазил в настройках, и цвета у вас на экране другие).

Константы

В качестве числовых констант можно использовать и шестнадцатеричные числа.

Шестнадцатеричное число состоит из шестнадцатеричных цифр, которым предшествует знак доллара $. Диапазон шестнадцатеричных чисел — от $00000000 до $FFFFFFFF.

Символьная константа — это любой символ, заключенный в апострофы:
' z' - символ z;

Если надо записать сам символ апострофа, он удваивается: ‘’’’ .

Строковая константа — любая последовательность символов (кроме символа возврата каретки), заключенная в апострофы. Если в строке нужно написать апостроф, - он удваивается, например: 'Helga''s Buch'.

3.2. Структура программы в Turbo Pascal

Структура программы в TP имеет следующий вид:

```
program    {Имя программы}
uses       {Список используемых модулей}
label      {Описание меток}
const      {Описание констант}
type       {Описание типов}
var        {Описание переменных}
procedure  {Блок описания процедур }
function   { Блок описания функций}
exports    {Описания экспортируемых имен}
           {Раздел операторов (Наша основная программа)}
begin  {начало исполняемой части}
{Операторы}
end.
```

Пока что не будем объяснять значения неизвестных для вас разделов, но заметим, что желательно описывать блоки программы именно в такой последовательности (хотя некоторые из них можно менять местами).

3.3. Числовые типы данных

В программах, написанных в предыдущей главе, мы использовали лишь 2 типа данных: Integer и Real. Но числовых типов в ТР в 5 раз больше: 5 целых типов и 5 вещественных. В таблицах приведены описания всех этих типов.

Целые типы

Идентификатор	Диапазон представления чисел	Требуемый размер памяти (в байтах)
Shortint	$-2^7..2^7-1$	1
Integer	$-2^{15}..2^{15}-1$	2
Longint	$-2^{31}..2^{31}-1$	4
Byte	$0..2^8-1$	1
Word	$0..2^{16}-1$	2

Размеры диапазонов прямо следуют из способа хранения целых чисел. Например, тип ShortInt – целый тип, принимающий как положительные, так и отрицательные значения, а значит, его двоичное представление записывается в двоичном дополнительном коде (см. п. 1.6.). Следовательно, минимальное отрицательное число записывается в виде: 1000 0000, или -128. А максимальное число, - в виде 0111 1111, или 127. Аналогичные рассуждения можно провести и для любого другого целого типа данных.

Вещественные типы

Давайте разберемся с представлением вещественного числа. Вы уже знаете, что действительное число печатается в виде $\pm mEp$. Например, 39.456, будет напечатано так: 3.9456000000E+01 (т.е. $3.9456 \cdot 10^1$). Число m называется мантиссой числа, p – показатель степени. Такое представление – «родное» для ПК, в котором любое вещественное число разбито на 3 части:

s	exp	m

s – знак числа (1 бит)
exp – показатель степени
m – мантисса (дробная часть)

Встроенные в ТР вещественные типы данных

Идентификатор	Диапазон представления чисел	Значащие цифры мантиссы	Требуемый размер памяти (в байтах)
Single	$1.5*10^{-45}..3.4*10^{38}$	7..8	4
Real	$2.9*10^{-39}..1.7*10^{38}$	11..12	6
Double	$5.0*10^{-324}..1.7*10^{308}$	15..16	8
Extended	$3.4*10^{-4932}..1.1*10^{4932}$	19..20	10
Comp	$-2^{63}+1..2^{63}-1$	19..20	8

Все вещественные типы, за исключением типа real, можно использовать лишь при подключении математического сопроцессора. Чтобы подключить мат. сопроцессор, надо перед началом программы объявить директиву {$N+}[7].

Не забывайте, что несмотря на то, что диапазон, например, чисел Real равен $1.5*10^{-45}..3.4*10^{38}$, это не означает, что все вещественные числа из этого диапазона могут быть записаны в разнесчастных 4 байтах[8], которые занимает переменная этого типа. Не забывайте, что 4 байтами можно задать лишь 2^{32} различных значений. Просто в мантиссе лишь первые 7-8 цифр будут точны, а все остальные могут быть какими угодно. Поэтому если вы захотите занести число в 100 знаков в тип extended то, несмотря на то, что диапазон значений у него огромный, но extended принципиально не может обеспечить того, чтобы младшие 80 знаков этого целого числа были нужного значения.

В ТР есть возможность печатать число не в стандартной записи, а используя специальный формат. Чтобы задать формат, надо после вещественного числа или вещественной переменной поставить :A:B, где A, B – целые числа, причем A – это количество символов, которые должны отводиться под число, а B – количество цифр в дробной части. Вторую часть формата можно не указывать.

Например, следующая строка

```
writeln(234.566:4:1);
```

напечатает число 234.566, округленное до десятых, т.е.

234.6.

Строка

```
writeln(234.566:4:5);
```

напечатает число, выделив под дробную часть 5 знаков:

234.56600

Заметьте, что первая часть формата была проигнорирована, т.к. она просто неверна и число не может быть умещено в 4 символа. Замечу, что первая часть формата (количество символов, которые отводятся под число), может использоваться и для целых чисел.

3.4. Ограничены ли вычислительные возможности компьютера?

Мы знаем, что:

- На любой тип данных существуют ограничения по величине числа. Для вещественных чисел есть, конечно, тип extended, которого должно хватить для практически любых расчетов. Но если надо, чтобы все вычисления проводились точно, то ограничение остается в силе.
- Числа вещественных типов на самом деле - рациональные, причем лишь их весьма небольшая часть.
- Иррациональные числа содержат бесконечное количество знаков после запятой, поэтому, например, число π нельзя точно записать в ПК (т.к. память – ограничена). Аналогичные рассуждения можно повторить для периодических дробей.

---

[7] Директивы компилятора будут рассмотрены в главе 5.

[8] Даже если бы у нас была бесконечная последовательность байтов $a_1, a_2, ..., a_n, ...$, то и их тоже бы не хватило, чтобы закодировать вещественные числа из любого сегмента.

Исходя из этих фактов, нередко заявляют, что в ПК невозможно представлять практически все числа и, следовательно, машинная арифметика имеет лишь ограниченное применение.

Давайте рассмотрим 3 вопроса:

1. Как описывать большие целые числа?
2. Как описывать периодические дроби?
3. Что делать с ограничением ПК в памяти и как представлять в ПК иррациональные числа?

1. Во-первых, большие числа можно легко задавать, используя массивы (глава 6). Тогда вы без проблем сможете работать с довольно большими числами, например, 10^{12332}. Именно задаче создания огромных чисел, с которыми при этом можно работать с неимоверной точностью и посвящен первый проект «Длинная арифметика».
2. С рациональными числами - еще проще: дробь – это совокупность двух целых чисел – числителя и знаменателя. Причем в их качестве можно использовать числа, построенные в 1-м пункте. В главе 9 («Записи и модули») мы напишем модуль, позволяющий работать с дробями.
3. Когда математик говорит об иррациональных числах то он не выписывает все число (величина листа бумаги тоже может считаться ограничением на «память»), а придумывает некоторое обозначение для него: $\sqrt{2}$, $\log_2 3$, π и т.д. И ничего не мешает придумать программисту любой тип данных, которых позволил бы реализовать, например, все радикалы любой степени вложенности. Разумеется, степень вложенности тоже ограничивается памятью, но этот довод я уже рассмотрел.

Единственное, что остается в защиту тезиса о «неполноценности машинной арифметики», - довод, который я назову «лингвистическим»: число можно описывать не только математическими символами, но и словами, например: «Пусть x – число, которое равняется количеству слов, которые используются в этой фразе, заключенной в кавычки, и которые имеют количество букв, большее 5» (т.е. x = 10). Это - серьезный довод. Можно научить компьютер распознавать определенные предложения, но суть тезиса в том, что на вход могут подаваться любые предложения. Этот тезис сводится к вопросу, может ли компьютер понимать прочитанный текст, или нет? Так как в определении могут фигурировать абсолютно любые слова и понятия, которые могут трактоваться по-разному в зависимости от контекста, то задать вопрос следовало бы так: возможен ли искусственный интеллект (ИИ)? Пока электронный мозг не создан, опровергнуть этот тезис полностью нельзя. А на сегодняшний день особых успехов на пути создания ИИ нет.

К вопросу о возможностях алгоритмических машин мы вернемся в главе 14.

3.5. Условный оператор

Вернемся от абстрактных рассуждений к делам будничным. Возможно, вы, тестируя программы, написанные вами в предыдущем разделе для разных начальных данных, заметили одну важную проблему: если, например, начальные значения катетов будут –3 и –4, то компьютер будет продолжать вычислять значения гипотенузы,

периметра и площади треугольника, не обращая внимание на то, что такие значения катетов недопустимы. Когда вы сами задаете значения катетов, используя константы, вы сами в ответе за правильность работы вашей программы. Однако, используя процедуру readln, мы не сможем гарантировать правильность ввода пользователем исходных данных, и программа выдаст неверный результат. Выход напрашивается сам собой: мы должны проверять введенные значения на правильность, и если они тест не прошли, программа должна уведомить пользователя о том, где была его ошибка. В TP, как и в других императивных языках, механизм проверки условий существует, и реализуется с помощью условного оператора if.

Синтаксис условного оператора:

```
if (условие) then
  оператор1;
```

Работает условный оператор так: сначала проверяется следующее за ним условие. Если условие выполняется, тогда выполняется оператор1, в противном случае оператор1 выполняться не будет.

Если внутри условного оператора надо выполнить несколько действий, то применяется так называемый составной оператор:

```
begin
оператор 1;
оператор 2;
. . . . .
оператор n;
end
```

Его действие состоит в следующем: он несколько операторов объединяет в один. Вид оператора if, если мы используем составной оператор, будет следующий:

```
if (условие) then
  begin
  оператор 1;
  оператор 2;
  . . . . .
  оператор n;
  end
```

Теперь можем модернизировать программу, вычисляющую конечную скорость и пройденное машиной расстояние, написанную нами на прошлом занятии.

Мы должны проверить входные данные на допустимость. Давайте рассмотрим, какие ограничения мы должны учесть. Мы упростим себе жизнь, и не будем учитывать ограничения на скорости и ускорения (хотя это стоило бы сделать, а не то пользователь может ввести такие начальные данные, при которых автомобиль сгорит в атмосфере). Учитывать будем лишь ограничение t>=0, т.е. нас будет интересовать положение автомобиля в будущем.

Пример 1: Программа, вычисляющая по v0, a, t скорость машины в конце пути и пройденное ею расстояние.

```
1 : uses CRT;
2 : var
3 :   S:real;
4 :   v:real;
5 :   v0:integer;
6 :   a:shortint;
7 :   t:integer;
8 : BEGIN
9 :   Clrscr;{Очищаем экран}
10:   write('Введите время движения машины (в сек.): ');
11:   read(t);
12:   write('Введите начальную скорость машины (в м/с): ');
13:   read(v0);
14:   write('Каково было ускорение (в м/(с*с))? ');
15:   read(a);
16:   if t<0 then
17:     begin
18:     writeln('Время не может быть <0');
19:     exit;
20:     end;
21:   v:=v0+a*t;
22:   S:=v0*t+(a*t*t/2);
23:   writeln('Скорость в конце пути равна ',v,' м/с');
24:   writeln('За время движения машина проехала ',S,' м');
25: END.
```

Отличия новой программы от старой:

- Время движения, начальная скорость и ускорение – переменные величины, которые вводит пользователь.
- В строке № 16 проверяется, правильно ли введено время. Мы предполагаем, что время отрицательным быть не может, поэтому если пользователь введет значение t<0, то программа должна выдать ему сообщение об ошибке и завершить свою работу. Так как должны быть выполнены 2 оператора, то надо использовать составной оператор. Выйти из программы можно с помощью процедуры exit. Если пользователь введет значение t>=0, то действия внутри оператора if выполняться не будут, и программа продолжит выполнение со строки 21.

Замечание: Если вам необходимо выполнять несколько операторов в теле оператора if, то не забывайте использовать составной оператор. Если вы напишете так:

```
if (условие) then
  оператор1;
  оператор2;
```

то оператор1 будет вложен в оператор if, а оператор2 к оператору if относиться не будет.

3.6. Условный оператор с ветвью else

Часто требуется выполнять одни действия при выполнении условия, а другие – при его невыполнении. Для того, чтобы программист мог реализовать эту возможность, к оператору if присоединяется ветвь else.

Синтаксис такого условного оператора будет следующим:

```
if условие then
  оператор1
else
  оператор2;
```

Замечание: Перед ключевым словом else “;” не ставится!!! Если “;” стоит, то компилятор выдает сообщение об ошибке (Error 113: Error in statement);

Алгоритм работы такого оператора следующий: проверяется условие, стоящее после ключевого слова if. Если оно выполняется, то выполняться будет оператор1, а оператор2 будет игнорироваться. В противном случае выполнятся будет оператор2, принадлежащий ветви else, а оператор1 выполняться не будет.

Если и основная ветвь условного оператора, и ветвь else содержат по несколько операторов, то условный оператор будет выглядеть так:

```
if (условие) then
  begin
  оператор1;
  оператор2;
  . . . . .
  операторn;
  end
else
  begin
  оператор1;
  оператор2;
  . . . . .
  операторn;
  end;
```

3.7. Делимость чисел. Операции div и mod

Вы уже знакомы с операцией деления ‘/’. Но если вы работаете с целыми числами, и вы хотите разделить одно число на другое, и при этом получить в результате другое целое число, то такая операция деления вам не подойдет, так как ее результат вещественный. Кроме того, часто при работе с целыми числами нужна операция нахождения остатка от деления одного целого числа на другое. Для решения этих проблем в TP существуют операции div и mod.

В этом пункте будем рассматривать только целые числа.

- Говорят, что число a делится на число b (обозначается это $a \vdots b$), если существует число c такое, что $a = bc$.
- Число, на которое делится число x, называется делителем числа x.
- Число a называется кратным числу b, если a делится на b.

- Если $a \vdots 2$, то число a называется четным. В противном случае a называется нечетным.

Пусть даны 2 числа a и b. Разделим столбиком число a на число b. Получим частное c и остаток от деления q. Тогда число a можно представить в виде $a = bc + q$.

Например:

```
-243|25
 225 9
  18
```

9 – частное от деления 243 на 25, 18 – остаток от деления, $243 = 25 \cdot 9 + 18$.

Операции нахождения частного и остатка от деления двух чисел, в ТР называются div и mod соответственно.
Например:

```
c:= 243 div 25;   {c:=9}
c:= 243 mod 25  {c:=18}
```

С помощью операции mod можно легко реализовать проверку делимости чисел:
$a \vdots b$ тогда и только тогда, когда (a mod b) =0.

В следующем примере пользователь вводит целое число, а компьютер отвечает, является ли оно четным или нечетным.

Пример 2: Проверка числа на четность.

```
1 : var
2 :   n:integer;
3 : BEGIN
4 :   writeln('Введите число');
5 :   readln(n);
6 :   if (n mod 2)=0 then {Если число делится на 2}
7 :     writeln('Число - четное')
8 :   else
9 :     writeln('Число - нечетное');
10:   readln;
11: End.
```

Если условие (n mod 2)=0 выполняется, то выводится сообщение «Число - четное». В противном случае выполняются операторы ветви else, т.е. выводится на экран сообщение «Число - нечетное».

3.8. Решение линейного уравнения

При решении уравнения $ax + b = 0$, где $a, b, x \in R$, возможны 3 случая:

1. $a \neq 0$, тогда есть один корень $x = -\dfrac{b}{a}$
2. $a = 0$, $b \neq 0$, тогда корней нет
3. $a = 0$, $b = 0$, то корни – все действительные числа.

Так как входные значения чисел a и b произвольные, то мы должны учесть все 3 случая.

Пример 3: Программа, которая решает линейное уравнение ax+b=0.

```
1 : Uses Crt;
2 : var
3 :   a,b:real;{Коэффициенты уравнения}
4 :   x:real;  {Корень, если он есть}
5 : BEGIN
6 :   writeln('Введите коэффициенты линейного уравнения (a, b)');
7 :   readln(a,b);
8 :   if a=0 then
9 :     if b=0 then {У 0x+0=0  бесконечное множество корней}
10:       writeln('У уравнения бесконечное количество корней')
11:     else       {Корней нет (получается ур-е 0x+b=0)}
12:       writeln('У уравнения корней нет')
13:   else
14:     begin
15:     x:=-b/a;
16:     writeln('Корень данного уравнения ',x);
17:     end;
18:   readln;
19: end.
```

С помощью процедуры readln можно вводить значения нескольких переменных, если перечислять их через запятую, как в строке 7. Тогда, чтобы ввести значения переменных, надо сначала ввести значение первой переменной, затем нажать Enter, потом значение второй переменной, и снова нажать Enter. Если число аргументов больше 2-х, поступать надо аналогично.

В строке 8 проверяется, равен ли коэффициент a нулю. Если да, то выполняется вложенный оператор, который тоже, в свою очередь, является условным, и проверяет, равен ли нулю коэффициент b. Если да, то выполняется строка 10, в противном случае – строка 12. После выполнения вложенного условия будет выполняться строка, которая будет первой после внешнего условного оператора, т.е. строка 18. Если же условие a=0 не выполняется, то управление передается на ветвь else внешнего условного оператора, т.е. на строку 14.

Замечание 1: ветвь else всегда присоединяется к ближайшему открытому условному оператору.

Замечание 2: Заметьте, что операторы, вложенные в любой из условных операторов, смещены на 2 пробела вправо. Это улучшает наглядность программы, в частности, можно легко понять, какая ветвь else к какому оператору if относится.

Для закрепления материала рассмотрим еще один пример такого же типа. В деталях реализации разберитесь сами.

Пример 4: Вычисление максимума трех чисел.

```
1 : uses Crt;
2 :
3 : var a,b,c:integer;
4 :     max:integer;
5 : begin
6 :   writeln('Введите три числа');
7 :   read(a,b,c);
8 :   if (a>b) then
```

```
9 :      if (a>c) then
10:        max:=a
11:      else    {если a>b и c>=a => c>b => c = max(a,b,c)}
12:        max:=c
13:    else
14:      if (b>c) then {если a<=b и b>c => b = max(a,b,c)}
15:        max:=b
16:      else
17:        max:=c; {если a<=b и b<=c => a<=c => c = max(a,b,c)}
18: writeln('Максимальное число из чисел ',a,' ',b,' ',c,' это
',max);
19: end.
```

3.9. Тип Boolean

Булевский (он же логический) тип – это тип данных, который может принимать лишь 2 значения: истина (true) и ложь (false). Фактически булевский тип является одним из перечисляемых типов (см. главу 10).

Неявно мы с вами уже неоднократно использовали булевский тип. Мы говорили: условный оператор проверяет значение выражения, стоящего между словами if и then и в зависимости от того, верно выражение или ложно, выполняются или операторы, следующие после then или после else. Но это не совсем точно: условный оператор ничего не должен проверять, т.к. выражение проверяется «само по себе»: любая операция сравнения принимает два аргумента (это могут быть не только числа), а возвращает результат булевского типа – либо true, либо false. Поэтому условный оператор на самом деле принимает значение булевского типа.

Например, пусть переменная b типа Boolean. Тогда будут допустимы следующие операторы присваивания:

```
b:=true;
b:=(34>45); {b:=false}
```

А следующая строка вызовет ошибку несоответствия типов (Type mismatch).

```
b:=(12*2);
```

Тип Boolean назван в честь английского математика Джорджа Буля – создателя булевой алгебры.

3.10. Логические операции

Логических операций в TP четыре:

- not - отрицание
- and – логическое и
- or - логическое или
- xor – исключающее или

Аргументы логических операций могут быть булевскими или целыми. Операция not является унарной (т.е. у нее один операнд). Остальные операции бинарные.

Будем считать, что значение true соответствует значению 1, a false – 0.

Следующая таблица приводит правила, по которым работают бинарные логические операции:

x	y	x and y	x or y	x xor y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Операция not работает так:
not(0)=1
not(1)=0

Если числа целые, то логические операции проводят побитовое применение логических операций, например:

```
not(00110010) = 11001101
00110010 and 01100110 = 00100010
00110010  or 01100110 = 01110110
00110010 xor 01100110 = 01010100
```

Пример 5: Использование логических операций.

```
1 : var
2 :   x,y:shortint;
3 :   b:boolean;
4 : begin
5 :   x:=6;  {0000 0110}
6 :   y:=5;  {0000 0101}
7 :   writeln(x and y);{0000 0100}
8 :   writeln(x xor y);{0000 0011}
9 :   writeln(x or y); {0000 0111}
10:   writeln(not(x)); {1111 1001}
11:
12:   readln;
13: end.
```

Замечание 2: Можно как угодно комбинировать логические операции, но вы должны помнить о приоритетах. Т.к. not – унарная операция, то ее приоритет – самый высокий. Затем следует операция and, затем – or и xor (у них одинаковые приоритеты).

В заключение темы, посвященной условным операторам, нам осталось рассмотреть один тонкий момент использования оператора if.

Вы знаете, что ветвь else присоединяется к ближайшему открытому условному оператору. Но иногда встречаются задачи, в которых надо присоединить ветвь else не к ближайшему оператору, а к одному из внешних операторов. Сейчас мы рассмотрим пример одной из них.

Пример 6. Представьте себе, что вы в воскресенье должны идти на курсы программирования. Вы вводите день недели (1-7), а программа должна выдать вам, куда вы должны сегодня идти: на программирование, или в школу.

```
1 : uses Crt;
2 :
3 : var
```

```
4 :   Tag:byte; {день}
5 : begin
6 :   ClrScr;
7 :   writeln('Введите день недели (1-7)');
8 :   readln(Tag);
9 :   if (Tag <1) or (Tag>7) then
10:     writeln('Где же вы в неделе столько дней найдете ');
11:
12: { Это - правильный фрагмент программы}
13:   if (Tag>=6) then
14:     begin
15:     if (Tag=7) then
16:       writeln('Все на программирование!!!')
17:     end
18:   else
19:     writeln('Снова в школу');
20:
21: {В следующих строках ошибка: ветвь else относится к вложенному
22:   оператору if, а не к внешнему, как нам надо.
23: }
24:   if (Tag>=6) then
25:     if (Tag=7) then
26:       writeln('Все на программирование!!!')
27:   else
28:     writeln('Снова в школу');
29: {
30: Предыдущие 5 строк, написанные в хорошем стиле, выглядели бы так
31:   if (Tag>=6) then
32:     if (Tag=7) then
33:       writeln('Все на программирование!!!')
34:     else
35:       writeln('Снова в школу');
36: }
37: end.
```

Посмотрите на оператор if в строках 13-19. Раньше мы с вами использовали составной оператор только если надо было выполнять внутри условного оператора несколько действий. Сейчас же мы использовали его для того, чтобы “закрыть” вложенный условный оператор. Теперь ближайшим открытым условным оператором будет внешний, и ветвь else будет присоединена именно к нему. Ниже (строки 24-28) приведено неправильное решение задачи. В том, что второй вариант работает неверно вы можете убедиться, если введете день от 1 до 5.

Задачи

1. Составить программу, определяющую результат гадания на ромашке («любит - не любит») по введенному количеству лепестков.
2. Написать программу, которая по введенному возрасту человека говорит, к какой возрастной группе человек относится: дошкольник, ученик, работник, пенсионер.
3. Перераспределить значения переменных x и y так, чтобы в x оказалось большее из этих значений, а в y – меньшее.
4. Дан рост трех человек. Определить, является ли их рост одинаковым.
5. Даны сторона квадрата и радиус круга. Определить, площадь какой фигуры больше.
6. Составить программу для вычисления значений заданных функций:
   - $f(x) = \begin{cases} x^2, x < 0 \\ x, x >= 0 \end{cases}$
   - $f(x) = \begin{cases} x^2 + 15e^x, x <= 1 \\ \dfrac{1}{\sqrt{15 + \ln x}}, x > 1 \end{cases}$
   - $f(x) = \begin{cases} \sin x + \dfrac{34}{x}, x \neq 0 \\ \dfrac{1}{\sqrt{15 + \{x\} \cdot x^4}}, x > 1 \end{cases}$
7. Известны площади квадрата и круга. Определить:
   - поместится ли квадрат в круге;
   - поместится ли круг в квадрате;
8. Дано трехзначное число. Найти сумму его цифр.
9. Дано трехзначное число. Узнать, какая из его цифр – наибольшая.
10. Вам дано уравнение $ax^2 + bx + c = 0$, $a \neq 0$. Выведите формулы, выражающие корни уравнения через дискриминант $D = b^2 - 4ac$.
11. Даны числа a, b, c. Найти вещественные корни уравнений:
    а) $ax^2 + bx + c = 0$ б) $ax^4 + bx^2 + c = 0$. Если корней нет, то сообщить об этом.
12. Даны произвольные числа a, b, c. Сказать, можно ли построить треугольник с такими длинами сторон, и если можно, то сказать, равнобедренный это треугольник, равносторонний, или общего вида.
13. Написать программу, которая вычисляет периметр и площадь правильного n-угольника, вписанного в окружность заданного радиуса.
14. **(!)** Вычислить значение производной функции $y = x^x$ в заданной точке a (a>0).
15. В подъезде жилого дома имеется n квартир, пронумерованных подряд, начиная с номера a. Определить, является ли сумма номеров всех квартир четным числом.
16. Выразить операцию and через операции not и or
17. Выразить операцию or через операции not и and
18. Выразить операцию xor через операции not, or, and.

Глава 4: Операторы циклов

На предыдущем занятии мы, вооружившись оператором if, сумели решить часто встречающуюся задачу нахождения минимума и максимума чисел. Но мы рассмотрели лишь частный случай: мы находили минимум 3 чисел. Давайте теперь рассмотрим более общую задачу: пользователь должен ввести некоторое количество чисел, количество которых не известно заранее, а программа должна выдавать их максимум. Используя лишь условные операторы, мы не сможем решить эту задачу.

На этом занятии вы изучите операторы циклов (for, while, repeat), с помощью которых вы сможете легко справляться с подобными задачами.

4.1. Цикл For

Существует 2 разновидности цикла for:

Цикл с инкрементом

```
for i:=k to m do
  begin
  оператор 1;
  оператор 2;
  . . . . . .
  оператор n;
  end;
```

Цикл с декрементом

```
for i:=k downto m do
  begin
  оператор 1;
  оператор 2;
  . . . . . .
  оператор n;
  end;
```

Разумеется, если в цикл вложен 1 оператор, то можно обойтись и без beginend'а.

Алгоритм работы цикла for с to:
1. Если k<=m, то выполняется оператор i:=k. Иначе – выход из цикла.
2. Выполняются операторы тела цикла.
3. Если k<m, то значение i увеличивается на 1 и затем переходим к шагу 2. Иначе – выход из цикла.

Алгоритм работы цикла for с downto:
1. Если k>=m, то выполняется оператор i:=k. Иначе – выход из цикла.
2. Выполняются операторы тела цикла.
3. Если k>m, то значение i уменьшается на 1 и затем переходим к шагу 2. Иначе – выход из цикла.

Замечание 1: в теле цикла for изменять значения счетчика не рекомендуется
Замечание 2: счетчик должен быть порядкового типа. Если тип счетчика вещественный, то компилятор выдает ошибку (Error 97: invalid FOR control variable).

Пример 1: Программа, выводящая числа 1..10 в возрастающем порядке, а затем 9..1 в убывающем.

```
1 : Uses Crt;
2 :
3 : var
```

```
4 :   i:integer;
5 : begin
6 :   ClrScr; {Очистка экрана}
7 :   writeln('Этот цикл будет печатать числа от 1 до 10');
8 :   for i:=1 to 10 do {этот цикл увеличивает значения i на 1}
9 :     writeln(i);     { пока i не станет равным 10}
10:   writeln('А этот - от 9 до 1 в обратном порядке');
11:   for i:=9 downto 1 do {этот цикл уменьшает значения i на 1}
12:     writeln(i);        {пока i не станет равным 1}
13: end.
```

Сначала запустите программу и посмотрите результат, а затем разберемся, как эта программа работает. До строки 8 ничего нового нет. А в строке 8 используется описанный выше оператор цикла for. Он работает так: как только начнет выполняться строка 8 нашей программы, счетчику i будет присвоена единица. После этого значение i будет сравнено с числом 10. Если i<10, то будут выполняться вложенные в цикл операторы (в нашем случае – вывод числа i на экран). После этого значение счетчика увеличивается на 1 и снова сравнивается с числом 10 и т.д. Когда будут выведены все числа от 1 до 10 на экран, значение переменной i будет 10 и, следовательно, цикл завершит свою работу и выполняться начнет оператор, следующий за оператором цикла (в данном случае – оператор в строке 10). Далее, в строке 11 используется еще один оператор цикла, но уже с использованием downto. Он будет работать так: переменной-счетчику будет присвоено значение 9, затем оно будет сравнено с 1. Если значение i будет больше, то цикл отработает, и выведет число i, т.е. число 9. Затем значение i уменьшится на 1, снова будет сравнено с 1 и т.д. Когда будут выведены все числа от 9 до 1, значение i будет равно 1 – значит, цикл завершит работу и начнет выполняться следующий оператор.

Среди программистов принято негласное соглашение называть счетчики буквами i, j, k.

4.2. Вычисление факториала

Пусть n - целое число.

- Число $n! = \begin{cases} 1\cdot 2\cdot \ldots \cdot n\ , n\ge 1 \\ 1, n=0 \end{cases}$ называется факториалом числа n.
- $(2n)!! = 2\cdot 4\cdot \ldots \cdot (2n-2)\cdot 2n$ – двойной факториал четного числа $2n$.
- $(2n+1)!! = 1\cdot 3\cdot \ldots \cdot (2n-1)\cdot (2n+1)$ – двойной факториал нечетного числа $2n+1$.

Пример 2: Программа, вычисляющая факториал числа.

```
1 : var i:byte;
2 :     n:integer;
3 :     f:longint;
4 : Begin
5 :   f:=1;
6 :   write('Введите число ');
7 :   readln(n);        {Вводим число n}
8 :   if n<0 then   {n должно быть >0 }
```

```
9 :     begin
10:     writeln('Вы ввели неверное число');
11:     exit;          {exit осуществляет выход из программы}
12:     end;
13:   {0!=1 и 1!=1}
14:   for i:=2 to n do  {последовательно умножаем f на числа
2,3,..n}
15:     f:=f*i;       {Если число n<2 => цикл выполняться не будет}
16:   writeln('  ',n,'! = ',f);
17: end.
```

В строке 5 переменной f, в которой мы будем хранить значение факториала, присваивается 1. Если мы этого не сделаем, то переменной f компилятор присвоит значение 0, и в дальнейшем, умножая число f на числа 2, 3, … n, мы получим в результате f=0.

Далее мы вводим число n, и проверяем, будет ли оно натуральным (факториалы отрицательных чисел мы не рассматриваем)

В строке 15 мы используем оператор цикла for для вычисления факториала числа. На первом витке цикла мы умножим число f на 2, затем на 3, и т.д., пока мы не умножим его на все числа от 2 до n. После этого значение счетчика i станет равным n, и цикл прекратит свою работу. В результате мы действительно получим значение n!.

Пример 3: Программа, находящая сумму наперед заданного количества чисел, введенных пользователем.

```
1 : const
2 :   n=4;{количество вводимых символов}
3 : var
4 :   i:byte;
5 :   x:integer;{переменная, в которую будем вводить числа}
6 :   S:longint;{Сумма чисел}
7 : begin
8 :   writeln('Введите ',n,' чисел');
9 :   for i:=1 to n do
10:     begin
11:     readln(x);{Вводим новое число (старое значение x теряется)}
12:     s:=s+x; {прибавляем новое число x к сумме предыдущих чисел}
13:     end;
14:   Writeln('Сумма чисел = ',S);
15: end.
```

В цикл (см. строку 9) вложены 2 оператора, поэтому надо обязательно использовать составной оператор (строки 10-13). На каждом из n витков цикла пользователь вводит с клавиатуры число x, а затем это число прибавляется к сумме предыдущих чисел. Заметьте, что при вводе нового значения x мы теряем предыдущее его значение. Но оно нам и не надо, т.к. цель нашей программы лишь найти сумму введенных чисел, поэтому перед вводом следующего числа нам достаточно хранить сумму предыдущих чисел, а не сами числа.

4.3. Циклы while и repeat

Теперь, когда вы умеете использовать цикл for, количество задач, которые вы можете решить, значительно возросло. Но у цикла for есть один недостаток – он применим лишь в случае, когда количество шагов заранее известно. Если это не так, то обычно используются операторы цикла while или repeat. Циклы for удобнее, если надо количество шагов заранее известно.

У цикла while следующий синтаксис:

```
while условие do
  begin
  оператор 1;
  оператор 2;
  . . . . . .
  оператор n;
  end;
```

Алгоритм работы цикла while:

1. Проверяется условие в заголовке.
2. **Если оно не выполняется, то выходим из цикла.** В противном случае выполняются операторы тела цикла. Затем программа переходит на начало цикла, снова проверяется условие и т.д.

Цикл repeat описывается так:

```
repeat
  оператор 1;
  оператор 2;
  . . . . . .
  оператор n;
until условие;
```

Алгоритм работы цикла repeat:

1. Выполняются операторы тела цикла.
2. Проверяется условие в конце цикла. **Если оно выполняется, то осуществляется выход из цикла**. Если же оно не выполняется, то переходим к пункту 1

Для того чтобы вам было легче сравнивать алгоритмы работы циклов for и while, в первом примере мы с помощью цикла while напишем программу вычисления факториала числа.

Пример 4: Программа, вычисляющая факториал числа с помощью цикла while.

```
1 : var i:byte;
2 :     n:integer;
3 :     f:longint;
4 : Begin
5 :   f:=1;
6 :   write('Введите число ');
7 :   readln(n);    {Вводим число n}
8 :   if n<0 then   {факториал для чисел n<0}
9 :     begin       {не определен}
10:     writeln('Вы ввели неверное число');
```

```
11:     exit;            {exit осуществляет выход из программы}
12:     end;
13:   i:=2;
14:   while i<=n do  {Этот цикл последовательно}
15:     begin        {умножает число f на числа 2,3,..n}
16:     f:=f*i;
17:     inc(i); {i:=i+1}
18:     end;
19:   writeln(' ',n,'! = ',f);
20: end.
```

В строке 13 значению счетчика i присваивается число 2. Затем в строке 14 проверяется условие, будет ли i<=n. Если да, то выполняются операторы тела цикла. Сначала число f умножается на число i, а затем число i увеличивается на 1 с помощью процедуры inc(i).

Замечание: эта процедура работает немного быстрее, чем i:=i+1, т.к. в языке Assembler предусмотрена специальная процедура для увеличения переменной на 1.

После этого программа возвращается к началу цикла, где снова проверяется условие и т.д.

Очевидно, что в этом примере мы с помощью цикла while смоделировали цикл for:

```
for i:=2 to n do
  f:=f*i;
```

Просто мы явно описали те действия, которые цикл for делает автоматически (т.е. проверку условия i<=n и увеличение счетчика на 1). Из этого следует, что цикл for является частным случаем цикла while. В принципе, можно внутри цикла for изменять значение счетчика и таким образом смоделировать while с помощью for. Поэтому на самом деле все операторы циклов эквивалентны.

4.4. Нахождение НОК и НОД 2-х чисел

В следующих определениях все числа подразумеваются целыми.

- Число x называется делителем числа y, если существует такое число c, что выполняется $y = cx$
- Общим делителем чисел x и y называется число, которое делится и на x, и на y.
- Наибольшим общим делителем чисел x и y (обозначается НОД(x, y), или просто (x, y)) называется наибольший из общих делителей чисел x, y.
- Натуральное число, которое делится на оба числа x, y называется общим кратным чисел x, y.
- Наименьшим общим кратным (обозначается НОК(x, y), или $\{x, y\}$, или $[x, y]$) называется наименьшее из общих кратных чисел x, y.

Например:

НОД(15,39)=3; НОК(15,39)=195
НОД(28,42)=14; НОК(28,42)=84

- Натуральное число >1 называется простым, если оно делится только на 1 и на самого себя.

Простые числа: 2, 3, 5, 7, 11, 13, 17, 19 …

- Натуральное число n называется составным, если у него существуют делители, отличные от 1 и n

Пример: 39 – составное (делители – 1, 3, 13, 39).

Искать НОД чисел бывает нужно в самых различных ситуациях. Простейшая из них – сокращение дробей.

Если числа малы, то найти их НОД и НОК можно методом подбора. Но что делать, если числа достаточно велики, скажем 121353 и 2954653. Тут уже метод подбора не пройдет. Нам нужны алгоритмы, которые позволят быстро находить НОД и НОК 2-х чисел.

Рассмотрим 2 из них:

Первый алгоритм базируется на использовании следующей теоремы, которую мы примем без доказательства.

Основная теорема арифметики:

Любое натуральное число a>1 единственным образом представляется в виде $a = p_1^{k_1} p_2^{k_2} ... p_n^{k_n}$, где $p_1 < p_2 < ... < p_n$ - простые числа, а $k_i, i = \overline{1,n}$ - целые положительные числа

Теперь пусть у нас есть 2 числа a, b. По основной теореме арифметике получим разложения (называемые каноническими)

$$a = p_1^{k_1} p_2^{k_2} ... p_n^{k_n}$$
$$b = t_1^{s_1} t_2^{s_2} ... t_m^{s_m}$$

Множества P=$\{p_1, p_2, ..., p_n\}$ и T=$\{t_1, t_2, ..., t_m\}$, вообще говоря, не совпадают (т.е. могут совпадать, а могут и нет).

Построим множество $Q = P \bigcup T$ - множество простых чисел, которые присутствуют в разложении хотя бы одного из чисел a, b.

Теперь разложим числа a, b по числам q_i:

$$a = q_1^{r_1} q_2^{r_2} ... q_f^{r_f}, \ q_j \in Q, \ r_j = \begin{cases} k_i, \exists i : k_i = r_j \\ 0, else \end{cases}$$

$$b = q_1^{c_1} q_2^{c_2} ... q_f^{c_f}, \ q_j \in Q, \ c_j = \begin{cases} s_i, \exists i : s_i = r_j \\ 0, else \end{cases}$$

Например:

Пусть $a = 180$, $b = 350$

$a = 2^2 \cdot 3^2 \cdot 5$, $b = 2 \cdot 5^2 \cdot 7$ - канонические разложения.

$a = 2^2 \cdot 3^2 \cdot 5 \cdot 7^0$, $b = 350 = 2 \cdot 3^0 \cdot 5^2 \cdot 7$ - разложения по числам из множества Q.

Теперь легко видеть, что

$$НОД(a,b) = q_1^{\min\{r_1,c_1\}} q_2^{\min\{r_2,c_2\}} ... q_f^{\min\{r_f,c_f\}} \quad (1)$$

$$НОК(a,b) = q_1^{\max\{r_1,c_1\}} q_2^{\max\{r_2,c_2\}} ... q_f^{\max\{r_f,c_f\}} \quad (2)$$

Под $\min\{a,b\}$ понимается минимальное из чисел a и b, а под $\max\{a,b\}$ - максимальное из чисел a и b.

Например, найдем НОД(1800,630):

$1800 = 2^3 3^2 5^2$

$630 = 2 \cdot 3^2 \cdot 5 \cdot 7$

$НОД(1800,630) = 2 \cdot 3^2 \cdot 5 = 90$

$НОК(1800,630) = 2^3 \cdot 3^2 \cdot 5^2 \cdot 7 = 12600$

Можно легко доказать (упражнение 16), что

$$a \cdot b = НОД(a,b) \cdot НОК(a,b) \quad (3)$$

Однако этот метод хорош скорее для теоретических построений, чем в качестве практического алгоритма. Задача разложения числа на простые множители – сложная задача, - гораздо сложнее, чем нахождение НОД (попробуйте разложить число 928734685 на простые множители). Хотелось бы вооружиться более простым способом решения нашей задачи. И такой способ есть. Называется он алгоритмом Евклида, и является одним из самых первых нетривиальных алгоритмов в истории науки.

Заключается он в следующем: пусть у нас есть 2 числа a и b.
Найдем q_1=a mod b - остаток от деления числа a на число b.

Тогда число a представимо в виде

$$a = c_1 b + q_1 \quad (4)$$

Если $q_1 \neq 0$, то делим число b на q_1, и получим новое выражение

$$b = c_2 q_1 + q_2 \quad (5)$$

Если $q_2 \neq 0$, то делим число q_1 на q_2

$$q_1 = c_3 q_2 + q_3$$

будем делить q_i на q_{i+1} пока $q_{i+1} \neq 0$

$$\ldots$$

$$q_{n-1} = c_{n+1} q_n + q_{n+1}$$

$$q_n = c_{n+2} q_{n+1}$$

Теперь докажем, что $НОД(a,b) = q_{n+1}$

Подставляя выражение $q_n = c_{n+2} q_{n+1}$ в выражение для q_{n-1}, получим, что $q_{n-1} \vdots q_{n+1}$.
Затем, подставляя выражения для q_{n-1} и q_n в равенство для q_{n-2}, получим, что $q_{n-2} \vdots q_{n+1}$.
И. т. д., поднимаясь по равенствам вверх, получим, что $q_i \vdots q_{n+1}, i = \overline{1,n}$, $b \vdots q_{n+1}, a \vdots q_{n+1}$

То есть число q_{n+1} - общий делитель чисел a, b.
Докажем теперь, что q_{n+1} - наибольший из общих делителей чисел a и b.
Предположим противное: пусть существует z – общий делитель чисел a и b: $z > q_{k+1}$. Из (4) $q_1 = a - c_1 b$. Так как $a \vdots z$ $b \vdots z$, то и $q_1 \vdots z$.

Аналогично из формулы (5) получим, что $q_2 \vdots z$.

Продолжая эти рассуждения, в итоге получим $q_{n+1} \vdots z$. Но по предположению $q_{n+1} < z$. Получили противоречие. Значит $q_{n+1} = НОД(a,b)$.

Следующая программа реализует алгоритм Евклида с помощью цикла while.

Пример 5: Программа, которая находит НОД и НОК 2 введенных чисел.

```
1 : Uses Crt;
2 :
3 : var
4 :   h,a,b,q:integer;
5 : begin
6 :   Clrscr;
```

```
7 :   writeln('Введите, пожалуйста, 2 числа');
8 :   readln(a,b);
9 :   h:=a*b;     {Произведение чисел a и b надо для вычисления НОК}
10:   q:=1;           {Если q=0, то цикл работать не будет}
11:   while q<>0 do
12:     begin
13:     q:=a mod b; {вычисляем остаток от деления чисел a и b}
14:     a:=b;
15:     b:=q;
16:     end;
17:   writeln('НОД чисел = ',a);
18:   h:=h div a;          {НОК(a,b) = a*b/НОД(a,b)}
19:   writeln('НОК чисел = ',h);
20:   readln;
21: end.
```

В начале программы мы сохраняем произведение чисел a и b (строка 9). Переменной q присваиваем число 1 только с одной целью: чтобы цикл начал свою работу. Вместо 1 можно было бы присвоить любое другое число, $\neq 0$.

В цикле while мы сначала вычисляем остаток от деления числа a число b. Но так как на следующем шаге цикла мы должны будем делить уже число b на число q, то надо сделать переприсваивание a:=b, b:=q. Затем переходим на новый виток цикла: проверяем условие q<>0. Если оно не выполняется, то начинается новый виток цикла: после вычисления остатка от деления a на b, на втором шаге цикла в переменной a будет записано число b, в переменной b число q_1, а в переменной q – число q_2. Затем снова делаем переприсваивание, и. т. д., пока на каком-либо шаге цикла остаток от деления не станет равным 0. Как только q=0, то мы выходим из цикла, а НОД(a,b) будет находиться в переменной a.

Эту же самую программу можно написать и с помощью цикла repeat (тогда она будет занимать на несколько строк меньше, и выглядеть несколько элегантнее):

Пример 6: Реализация алгоритма Евклида с помощью цикла repeat.

```
1 : Uses Crt;
2 :
3 : var
4 :   h,a,b,q:longint;
5 : begin
6 :   Clrscr;
7 :   writeln('Введите, пожалуйста, 2 числа');
8 :   readln(a,b);
9 :   h:=a*b;     {Произведение чисел a и b надо для вычисления НОК}
10:   repeat
11:     q:=a mod b; {вычисляем остаток от деления чисел a и b}
12:     a:=b;
13:     b:=q;
14:   until q=0;
15:   writeln('НОД чисел = ',a);
16:   h:=h div a;               {НОК(a,b) = a*b/НОД(a,b)}
17:   writeln('НОК чисел = ',h);
```

```
18:   readln;
19: end.
```

Реализация получилась немного короче, т.к. не понадобилось инициализировать число q начальным значением. Однако так бывает далеко не всегда, поэтому при написании конкретной программы сам программист выбирает, какой цикл подходит больше.

4.5. Процедуры break и continue

Процедура continue вызывает пропуск оставшейся части цикла, и начинается выполнение следующей итерации цикла.

Пример 7: Использование процедуры continue.

```
1 : var
2 :   n:integer;
3 :   spos,sneg:integer;
4 :
5 : begin
6 :   spos:=0; {Начальное количество положительных чисел равно 0}
7 :   sneg:=0; {Начальное количество отрицательных чисел равно 0}
8 :   repeat
9 :     read(n);     {вводим новое число}
10:     if n>0 then  {Если число положительно}
11:       begin
12:       inc(spos); {spos:=spos+1}
13:       continue;  {Переходит на новую итерацию цикла}
14:       end;
15:     if n<0 then  {Если число отрицательно}
16:       inc(sneg); {sneg:=sneg+1}
17:   until n=0;    {Если число=0 то цикл завершает свою работу}
18:   writeln(' Количество положительных чисел = ',spos);
19:   writeln(' Количество отрицательных чисел = ',sneg);
20: end.
```

Когда программа дойдет до слова repeat, начинается выполнение тела цикла (предварительных условий нет!): пользователь вводит число, и если оно положительно, то выполняется тело цикла (строка 10): кол-во положительных чисел увеличивается на 1, и затем с помощью процедуры continue программа переходит на начало цикла (второй условный оператор не выполняется).

С помощью процедуры break можно выходить из цикла. После выполнения процедуры break программа продолжит свою работу с первого оператора после оператора цикла, в котором находится процедура break. Но запомните: процедура break выходит только из одного оператора цикла!!!

Например, в следующем фрагменте программы после процедуры break будет выполняться процедура inc(d);

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

```
    break;
    s:=1;
    read(h);
  until h=1;
  inc(d);
  end;
s:=sqr(h);
```

4.6. Проверка чисел на простоту

Простые числа интересны не только с точки зрения чистой математики. Разложение больших чисел (около 200 десятичных разрядов) на простые множители, если сами простые числа, участвующие в разложении, также велики, - очень трудная задача, и на данный момент не существует алгоритмов, позволяющих быстро выполнять разложение. Поэтому на основе этой проблемы строятся многие криптосистемы.

Простейший способ проверки числа на простоту – делить его на все числа, меньшие его. Если число разделится хоть на одно из этих чисел, то число – составное. Проверка простоты числа занимает много времени, поэтому желательно уменьшить область перебора делителей. Первый шаг в этом направлении – элементарен: для проверки на простоту числа n можно делить лишь на числа, $\le \left[\frac{n}{2}\right]$.

Следующая теорема дает более сильный результат.

Теорема: минимальный простой делитель составного числа n не превосходит $\sqrt{n}$

Доказательство:

Пусть минимальный простой делитель числа n равен a. Предположим, что теорема не верна, т.е. $a > \sqrt{n}$. Так как n составное, то у него есть еще по крайней мере один делитель, не равный 1. Следовательно, $n \ge a \cdot a > \sqrt{n} \cdot \sqrt{n} = n$. получили противоречие. Следовательно, $a \le \sqrt{n}$.

Значит, чтобы проверить число n на простоту, мы может делить его на все числа $\le \sqrt{n}$. Если мы нашли хоть одно число, на которое число n разделится, то n – составное. Следующая программа реализует этот алгоритм.

Пример 8: Программа, печатающая простые числа в промежутке [2,n].

```
1 : Uses Crt;
2 : var
3 :   i,k,j,n:integer;
4 : begin
5 :   Clrscr;
6 :   write('Введите число ');
7 :   readln(n);
8 :   write(2,' ');
9 :   for i:=3 to n do  {Просматриваем все числа}
10:     begin           {и проверяем их на простоту}
11:     k:=round(sqrt(i)); {Наименьший простой делитель числа x
12:                         лежит в промежутке [2,sqrt(x)]}
13:     for j:=2 to k do      {Делим i на числа 2..k}
```

```
14:         begin
15:         if (i mod j)=0 then {Если число i делится на j}
16:           break;            {то i - не простое число}
17:         if j=k then         {Если i не делится ни на одно из}
18:           write(i,' ');     {чисел 2..k, то i - простое}
19:         end;
20:       end;
21:   readln;
22: end.
```

Первое простое число – 2. Мы его выводим отдельно от остальных (дело в том, что даже если мы начнем внешний цикл for с 2, а не с 3, то наша программа не выведет число 2 на экран (подумайте, почему)).

После того, как мы напечатали число 2, мы должны поочередно проверить на простоту все числа от 3 до n. Внутри внешнего цикла мы вычисляем $\sqrt{i}$, а затем мы должны проверить, делится ли число i хотя бы на одно из чисел промежутка $[2,\sqrt{i}\,]$. Эту проверку мы выполняем в строке 15. Если мы нашли такое число j, что $i \vdots j$, то мы должны выйти из цикла проверки числа на простоту, что и делает процедура break. Если же число i не разделилось ни на одно из чисел промежутка $[2,\sqrt{i}\,]$ (проверка этого условия – в строке 17), то оно – простое, и мы должны его напечатать.

4.7. Как компьютер может вычислить значение функции?

Это вопрос не такой простой, каким он кажется на первый взгляд. На аппаратном уровне компьютер может лишь складывать и умножать целые и вещественные числа, а также вычислять остаток и неполное частное от деления двух целых чисел (т.е. операции mod, div). Никаких степенных функций, логарифмов и синусов компьютер сам по себе не вычисляет. Его надо научить это делать.

Какую функцию реализовать проще всего? – Естественно, возведение в натуральную степень. Чтобы вычислить x^n достаточно n раз умножить число x само на себя. Можно возведение в степень реализовать намного лучше, но дело не в этом. Главное – что с помощью ЭВМ можно возводить число в натуральную и, следовательно, в целую степень.

С другими функциями дело обстоит сложнее. Однако есть универсальный способ, основанный на представлении функции в виде бесконечночлена (математики называют его степенным рядом). Например: $e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \ldots + \frac{x^n}{n!} + \ldots$, причем это верно для любых действительных чисел (иногда можно получить равенства, которые верны лишь на некотором интервале).

Я не буду останавливаться на доказательстве этого утверждения, поэтому тем, кто не знает основ матанализа, придется поверить, что это так. Более того, все другие элементарные функции тоже можно представить в виде рядов. Поэтому мы можем задачу приближенного вычисления значений элементарных функций свести к задаче приближенного вычисления бесконечночлена с некоторыми коэффициентами. Этим мы и займемся.

Приближенное вычисление бесконечных сумм

Для того, чтобы вычислить сумму членов произвольной конечной последовательности, достаточно просто просуммировать их. Гораздо сложнее дело обстоит с бесконечными суммами: в некоторых частных случаях, когда последовательность строится по определенному закону, можно вывести формулу, которая позволит точно вычислить сумму. Классическим примером является бесконечная убывающая геометрическая прогрессия, сумму элементов которой мы нашли в главе 1. Однако другие бесконечные суммы (математики говорят: суммы рядов) подсчитать не так просто, а в большинстве случаев и невозможно. Поэтому применяются приближенные формулы вычисления сумм рядов.

Итак, нам надо вычислить сумму $S = b_1 + b_2 + ... + b_n + ...$

Как можно поступить: мы будем находить так называемые усеченные суммы $S_n = b_1 + b_2 + ... + b_n$. Сначала найдем S_1, затем S_2, затем S_3 и т.д. А прибавлять каждый следующий член последовательности b_n будем только если $|b_n| > \varepsilon$, где ε - число, которое задается в начале вычислений, и называется точностью вычислений. Фактически мы считаем, что если прибавление каждого следующего числа практически не изменит результата, то заданная точность вычислений достигнута, и вычисления можно прекращать.

Чтобы наш метод давал верные результаты, мы будем предполагать, что $|b_{n+i}| < |b_n|$ для любого натурального числа i, а также что сумма конечна и последовательность убывает «достаточно быстро» (например, так как геометрическая прогрессия или еще быстрее). Если хотя бы одно из этих условий будет нарушено, то наш метод может не дать верного результата.

Например, сумма $1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n} + ...$ - бесконечна, хотя ее слагаемые образуют монотонно убывающую к нулю последовательность.

Теперь рассмотрим конкретный пример: найдем сумму $1 + \frac{x}{1!} + \frac{x^2}{2!} + ... + \frac{x^n}{n!} + ...$ для любого наперед заданного числа x. Т.к. мы знаем, что этот бесконечночлен не что иное, как экспонента, то эта сумма конечна. Кроме того, как только выполнится $|x| < n$, то каждое следующее слагаемое суммы будет по модулю меньше предыдущего, и убывание будет обратно факториалу, что очень быстро (быстрее, чем геометрическая прогрессия). Значит, приближенный метод вполне применим.

Напишем теперь код самой программы:

Пример 9: Приближенное вычисление суммы ряда.

```
1 : var
2 :   i:integer;
3 :   x,f,rat:real;
4 :   eps:real;{Точность вычислений}
5 : begin
6 :   writeln('Введите число x');
7 :   readln(x);
8 :   writeln('Введите точность вычислений');
9 :   readln(eps);
10:   {Присваиваем начальные значения вспомогательным переменным}
```

```
11:   f:=1;
12:   rat:=1;
13:   i:=1;
14:   repeat
15:     rat:=rat*x/i;{Вычисляем значение следующего слагаемого}
16:     inc(i);
17:     f:=f+rat;   {Прибавляем новый элемент ряда к сумме}
18:   until abs(rat)<eps; {Если модуль слагаемого меньше eps,}
19:                       {прекращаем вычисления}
20:   writeln('Результат: ','exp(',x,') = ',f);
21:   writeln('Результат процедуры, встроенной в TP ',exp(x));
22: end.
```

В данном примере число i – это номер слагаемого, которое вычисляется на данном витке цикла, rat – само i-е слагаемое, а f – сумма последовательности.

На первом шаге i=1, rat=1, f=1.

Так как, $\frac{x^{n+1}}{(n+1)!}=\frac{x}{n+1}\frac{x^n}{n!}$, то каждое следующее слагаемое можно получить из предыдущего, умножая на x, и деля на номер слагаемого (строка 15). Поэтому мы можем обойтись без трудоемкой процедуры возведения числа в степень. В 16-й строке увеличиваем i, так как на следующем шаге мы будем вычислять следующий член последовательности. В 17-й строке мы прибавляем текущее слагаемое к сумме, и затем переходим на следующий виток цикла, если не выполняется условие в строке 18.

Сравните результаты наших с вами вычислений с результатом, который дает встроенная функция. Вы увидите, что мы все сделали правильно.

Итак: приближенное вычисление любой элементарной функции может быть сведено к операциям сложения, умножения и деления; следовательно, компьютер может вычислить их значение в любой точке за конечное время с любой наперед заданной степенью точности.

В предыдущем утверждении очень важна фраза «с любой наперед заданной степенью точности». Дело в том, что если сумма $a_1+a_2+...+a_n+...$ конечна, то для любого $\varepsilon>0$ существует k такое, что $a_{k+1}+a_{k+2}+...<\varepsilon$. Однако каким будет это число k - 10, 1000 или 10^{10} - в общем случае сказать нельзя. Если бы мы начали вычислять $S_n=a_1+a_2+...+a_n$ для всех n, то мы на каком-то шаге непременно получим значения искомой суммы с любой степенью точности. Но можно подобрать такие суммы, что если мы зададим наперед точность ε, то мы не сможем узнать, когда нам надо приостанавливать вычисления - через минуту, день, год… Мы можем уже получить приближенное значение суммы, однако не сможем доказать, что оно таковым является. А если утверждается, что мы можем вычислить сумму с любой наперед заданной степенью точности, то это означает, что мы можем не только подсчитать приближенное значение суммы, но и доказать что оно таковым является и, следовательно, что мы можем прекращать дальнейшие вычисления.

Задачи

1. Написать программу вычисления факториала с помощью цикла repeat.
2. Вычислить сумму $S=1^2+2^2+...+n^2$ для наперед заданного числа n.

3. Логической переменной x присвоить значение true или false, в зависимости от того, является натуральное число k степенью 3 или нет.
4. Дано 8 вещественных чисел. Вычислить разность между минимальным и максимальным из них.
5. Дана непустая последовательность различных натуральных чисел, за которой следует 0. Определить порядковый номер наименьшего из них.
6. Дано число x. Придумать как можно более эффективный алгоритм и вычислить:

а) $y = x^{10} + 2x^9 + ... + 10x + 11;$

б) $y = 11x^{10} + 10x^9 + ... + 2x + 1;$

Даже если вы быстро решите эту задачу, все равно не поленитесь посмотреть в ответы, для того, чтобы убедиться в том, что вы нашли оптимальный алгоритм решения этой задачи.

7. Вычислить $y = 1! + 2! + 3! + ... + n!$ (n>0);
8. Числа Фибоначчи (f_n) определяются формулами

$f_0 = f_1 = 1;$ $f_n = f_{n-1} + f_{n-2}$ при $n = 2, 3, \dots$

a) определить 40-е число Фибоначчи;

b) найти первое число Фибоначчи, большее m (m>1);

c) Вычислить s – сумму всех чисел Фибоначчи, которые не превосходят 1000.

9. Не используя стандартные функции (за исключением abs), вычислить с точностью eps>0:

$$y = shx = x + \frac{x^3}{3!} + \frac{x^5}{5!} + ... + \frac{x^{2n+1}}{(2n+1)!} + ...$$

Считать, что необходимая точность достигнута, если очередное слагаемое по модулю меньше eps, - все последующие слагаемые можно уже не учитывать.

10. Дана непустая последовательность ненулевых целых чисел, за которой следует 0. Определить, сколько раз в этой последовательности меняется знак (например, в последовательности 1, -3, 4, 55, -4 знак меняется 3 раза).
11. Определить, является ли данное натуральное число совершенным, т.е. равным сумме всех своих положительных делителей, кроме самого себя (например, число 6 – совершенно, т.к. 1+2+3=6).
12. Составить программу печати таблицы температур по Цельсию от 0 до 100 градусов с шагом в один градус и их эквивалентов по шкале Фаренгейта, используя для перевода формулу $t_F = \frac{9}{5} t_C + 32$
13. В водоеме n тонн рыбы. Каждый год рыболовецкая бригада вылавливает x тонн. Воспроизводство рыбы – y % в год. Для сохранения воспроизводства нужно прекращать лов, когда в водоеме останется z тонн рыбы. Через сколько лет надо прекращать лов рыбы. Для простоты будем считать, что прирост популяции происходит мгновенно в конце года.
14. Дан прямоугольник размера $a \cdot b$. На первом шаге от него отсекают квадрат максимального размера, затем к оставшемуся прямоугольнику применяют аналогичную процедуру. И так до того момента, пока длина стороны отсекаемого квадрата не будет меньше c. Вычислить площадь прямоугольника, который остается от исходного после выполнения этой процедуры.
15. Дано натуральное число n. Найти сумму его цифр.
16. Докажите формулу $ab = НОД(a,b) \cdot НОК(a,b)$

17. $НОД(x_1, x_2, ..., x_n)$ назовем наибольшее из чисел, на которые делятся все x_i, $i = \overline{1,n}$. Напишите программу, которая будет считать НОД n введенных натуральных чисел.
18. Проверьте, можно ли обобщить формулу из упражнения 16 следующим образом: $x_1 x_2 ... x_n = НОД(x_1, x_2, ..., x_n) \cdot НОК(x_1, x_2, ..., x_n)$
19. Напечатать все простые делители заданного натурального числа.
20. Дана последовательность положительных вещественных чисел $x_1, x_2, ..., x_n$ (n заранее не известно), содержащая по крайней мере 1 число, за которыми следует отрицательное число. Вычислить величину $nx_1 + (n-1)x_2 + ... + 2x_{n-1} + x_n$.
21. Если внимательно присмотреться к формулам

$1 + 2 + ... + n = \frac{n(n+1)}{2}$ и $1^2 + 2^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6}$, то можно увидеть закономерность: в правых частях обеих формул стоят многочлены от переменной n, со степенями на 1 выше, чем степени чисел, которые мы суммируем.

Обобщите результат, показав, что суммой $1^p + 2^p + ... + n^p$ будем многочлен ($p+1$)-й степени.

22. Пользуясь результатом предыдущей задачи, найдите выражение для суммы $1^p + 2^p + ... + n^p$.
23. Вводится число n>0. Найти, сколькими нулями оканчивается $n!$. Число $n!$ может быть таким большим, что не влезет в любой стандартный числовой тип ТР.
24. Пусть выражение из цифр получается последовательным выписыванием всех натуральных чисел: 1234567891011121314151617181920… По заданному числу k выведите k-ю цифру последовательности.
25. Пусть последовательность из цифр получается так же, как и в задаче 24. Пусть $f(n) = m$, если 10^n-тая цифра выражения входит в состав m-значного числа. Например, $f(2) = 2$, так как сотая цифра входит в состав двузначного числа 55. Найдите (с доказательством) $f(1987)$.
26. Обобщите формулы, полученные для сумм арифметической и геометрической прогрессии, подсчитав сумму первых n членов следующей последовательности: $a_{n+1} = qa_n + d$.
27. В первой части было доказано, что $\sqrt{2}$ - иррациональное число. Докажите:
    - Если число p - простое, а $k > 1$, $k \in N$, $l < k$, то $\sqrt[k]{p^l}$ - иррациональное.
    - Используя основную теорему арифметики, исследуйте более общий случай: при каких значениях иррациональным будет число $\sqrt[p]{n}$, $n, p \in N$.
28. Пусть вам даны канонические разложения целых чисел a, b. Исследуйте, при каком условии число $\log_a b$ будет иррациональным.

Глава 5: Несколько операторов на закуску

В этой главе мы рассмотрим еще несколько операторов и типов данных, встроенных в TP а также обсудим, что мы уже знаем о программировании.

5.1. Оператор case

С помощью оператора if можно проверять любое условие, поэтому введение дополнительного оператора выглядит несколько искусственным. Но в ряде случаев использование оператора if не очень удобно, поэтому для упрощения программного кода был введен оператор case.

Синтаксис:

```
case выражение of
список значений 1: begin
                   операторы;
                    end;
список значений 2: begin
                   операторы;
                    end;
  . . . . . . . . . . . .
список значений n: begin
                   операторы;
                    end;
else
  begin
  операторы;
  end;
end;
```

Список значений – это набор констант, разделенных запятыми и диапазонов чисел (например 1,2,5,7..15,18..29)

Алгоритм работы:

1. Вычисляется значение выражения, следующего за словом case
2. Полученное значение последовательно сравнивается с константами из списков констант перед двоеточием.
3. Если значение нашего выражения попало в какой-то список констант, то выполняются операторы, следующие за двоеточием, и затем оператор case завершается.
4. Если значение выражения не находится ни в одном списке констант, то выполняются операторы ветви else (если она в операторе есть). Если же ее нет, то оператор case заканчивает свою работу.

Пример 1: Определение дня недели.

```
1 : var
2 :   i:integer;
3 : begin
4 :   writeln('Введите день недели (1-7)');
5 :   readln(i);
```

```
6 :   case i of
7 :   1..5:writeln('Рабочий день');
8 :   6:writeln('Суббота');
9 :   7:writeln('Воскресенье');
10:   else
11:     writeln('Вы ввели неверный день недели');
12:   end;
13:   readln;
14: end.
```

Заметьте, что оператор case должен обязательно заканчиваться словом end (открывающий begin не нужен). Аналогичная программа, написанная с помощью оператора if, выглядит так:

Пример 2: case против if.

```
1 : var
2 :   i:integer;
3 : begin
4 :   writeln('Введите день недели (1-7)');
5 :   readln(i);
6 :
7 :   if (i>=1) and (i<=5) then
8 :     writeln('Рабочий день')
9 :   else
10:     if (i=6) then
11:       writeln('Суббота')
12:     else
13:       if (i=7) then
14:         writeln('Воскресенье')
15:       else
16:         writeln('Вы ввели неверный день недели');
17:   readln;
18: end.
```

Очевидно, что с помощью оператора case программа выглядит гораздо понятнее.

5.2. Символьный тип

Тип char занимает в памяти 1 байт. С помощью одного байта можно закодировать 256 различных чисел. Для того чтобы закодировать символы, надо каждому из целых чисел от 0 до 255 поставить в соответствие определенный символ. Кодировки могут быть различны, поэтому для того, чтобы все программы могли стыковаться друг с другом, требуется ввести некоторую стандартную кодировку.

Стандартной кодировкой (т.е. соответствием между символами и числами) является кодировка ASCII (American Standard Code for Information Interchange – американский стандартный код для обмена информацией). Это 7-битный двоичный код, с помощью которого можно закодировать 128 чисел в диапазоне от 0 до 127.

Например, символу ‘a’ (латинской букве a) ставится в соответствие число 97, большой латинской букве ‘A’ – число 65 и т.д.

Вторая половина символов с кодами от 128 до 255 - может меняться на PC разных типов. Эти символы могут использоваться для кодирования специальных символов и букв других алфавитов, например, кириллицы.

Переменные символьного типа можно присваивать друг другу, вводить с помощью read, readln, а выводить с помощью write, writeln.

Подпрограммы для работы с типом char

Ord(ch)	Возвращает ASCII-код символа ch.
Chr(n)	Возвращает символ, код которого – число n типа byte.
Pred(ch)	Возвращает символ, предшествующий символу ch.
Succ(ch)	Возвращает символ, следующий за символом ch.

Замечание: не путайте символы с числами: символ ‘1’ и число 1 имеют разное машинное представление.

Пример 3: Демонстрация работы встроенных функций pred, ord, succ, chr.

```
1 : var
2 :   c:char;
3 :   n:byte;
4 : BEGIN
5 :   writeln('введите символ');
6 :   readln(c);
7 :   writeln('ASCII-код символа c = ',ord(c));
8 :   writeln('Символ, стоящий перед ',c,' - ',pred(c));
9 :   writeln('Символ, стоящий после ',c,' - ',succ(c));
10:   writeln('Введите код символа');
11:   readln(n);
12:   writeln('Символ, код которого n: ',chr(n));
13: end.
```

Пример 4: Вывод на экран всей таблицы ASCII.

```
1 : uses Crt;
2 : var
3 :   i:integer;
4 : begin
5 :   Clrscr;
6 :   for i:=0 to 255 do
7 :     write(chr(i));
8 :   readkey;
9 : end.
```

5.3. Работа с клавиатурой

Есть специальная область памяти, называемая буфером клавиатуры, в котором может храниться до 127 символов. Когда этот буфер переполняется, то динамик начинает издавать характерное звучание. Если вы когда-либо включали компьютер, а в это время на клавиатуре лежала книга, то вы его наверняка слышали. Если нет, то вы можете этот эксперимент провести в гораздо менее экстремальных условиях: просто, когда вам предлагают ввести символ с помощью readln, то зажмите любую клавишу, и

после того, как на экране появится 127-й символ, дальнейшее отображение символов на экране прекратится, а вместо этого динамик начнет издавать терзающие его динамическую душу звуки.

В модуле Crt есть полезная функция Readkey, которая возвращает первое число из буфера клавиатуры, если он не пуст и ждет нажатия клавиши в противном случае. При этом вводимые символы на экране не отображаются, что очень удобно, например, при использовании пароля.

Для перевода букв в верхний регистр используется функция UpCase(ch). Если символ ch – латинская буква, то функция вернет заглавную букву. Любой другой символ (в том числе и буквы кириллицы) функция изменять не будет.

Все клавиши и их комбинации можно разбить на 3 группы:

1. Клавиши и комбинации клавиш, которые посылают в буфер клавиатуры ASCII-код (буквенные и цифровые клавиши, а также комбинации Ctrl+некоторые другие клавиши).
2. Клавиши и комбинации клавиш, которые посылают в буфер клавиатуры расширенный код (F1..F12, комбинации Ctrl, Alt, Shift, с клавишами F1..F12, а также некоторые комбинации Alt+другая клавиша).
3. Клавиши и комбинации клавиш, не посылающие в буфер клавиатуры коды (Shift, Ctrl, Alt, NumLock, CapsLock, ScrollLock и некоторые комбинации клавиш).

Расширенный код состоит из символа 0 и следующего за ним кода клавиши.

Пример 5: Программа, которая по нажатию клавиши выдает ее код.

```
1 : Uses Crt;
2 : var
3 :   c: char;
4 : begin
5 :   repeat
6 :     c:=ReadKey;{Ждем нажатия клавиши}
7 :     if ord(c)<>0 then{Если клавиша посылает ASCII-код}
8 :       writeln(ord(c))
9 :     else    {Если клавиша посылает расширенный код}
10:       writeln('0',ord(Readkey):8)
11:   until ord(c)=27 {27 - ASCII-код клавиши Esc}
12: end.
```

В этой программе мы на каждом витке цикла просим пользователя нажать клавишу. После этого программа анализирует, посылает ли данная клавиша ASCII-код или расширенный код (если нажата клавиша, не посылающая код, то программа просто пропустит ее). Если клавиша посылает расширенный код, то выводится сначала 0, а затем второе число расширенного кода (строка 10).

Давайте подробнее рассмотрим случай, когда нажатие на клавишу посылает расширенный код. В этом случае в буфер клавиатуры посылаются 2 числа: 0 и второе число. При этом в переменную c (см. строку 6) будет записан лишь символ с кодом 0, а второй символ (с порядковым номером, равным второй части кода) останется в буфере клавиатуры. Для того чтобы его извлечь оттуда, надо применить еще раз функцию Readkey. Чтобы выйти из программы, надо нажать клавишу Esc (ASCII-код – 27).

Следующая программа отображает все вводимые пользователем строчные латинские буквы как заглавные. Ввод символов можно прекратить, нажав клавишу 1.

Пример 6: Отображение букв в верхнем регистре.

```
1 : uses Crt;
2 : var
3 :   c:char;
4 : begin
5 :   while c<>'1' do
6 :     begin
7 :     c:=readkey;
8 :     writeln(UpCase(c));
9 :     end;
10: end.
```

5.4. Генератор псевдослучайных чисел

В программах часто надо иметь возможность генерировать случайные числа. ПК, однако, не может генерировать полностью случайные числа, поэтому используются специальные алгоритмы, при помощи которых достигаются примерно одинаковые вероятности выпадения любого числа из заданного диапазона. В идеале вероятности появления всех чисел должны быть равны. В TP есть встроенная функция, генерирующая псевдослучайные числа.

Random(n) - Возвращает целое псевдослучайное число от 0 до n-1.

Random - Возвращает вещественное псевдослучайное число от 0 до 1.

Также вам понадобится процедура **Randomize**, которая инициализирует генератор псевдослучайных чисел случайным значением от системного таймера. Это число записывается в предопределенной переменной **RandSeed**.

Если перед началом работы с генератором случайных чисел (ГСЧ) не вызвать процедуру **Randomize**, то после каждого запуска программы вы будете получать те же значения случайных чисел.

В следующем примере вы увидите, насколько случайные числа выдаются встроенным генератором. мы заставим ГСЧ 1000000 раз выдать случайное число от 0 до 2, и подсчитаем, сколько раз он выдал каждое из этих чисел. Т.к. количество испытаний велико, то сильных отклонений от равномерного распределения быть не должно.

Пример 7: Использование генератора псевдослучайных чисел.

```
1 : const
2 :   n=1000000;
3 : var
4 :   i,x,y,z:longint;
5 :
6 : begin
7 :   randomize;
8 :   for i:=1 to n do
9 :     case random(3) of
10:     0:inc(x);
```

```
11:      1:inc(y);
12:      2:inc(z);
13:      end;
14:    writeln('Количество испытаний ',n);
15:    writeln('Ноль выпал ',x,' раз');
16:    writeln('Единица выпала ',y,' раз');
17:    writeln('Двойка выпала ',z,' раз');
18:    readln;
19: end.
```

Рассмотрим еще один пример использования процедуры randomize.

Пример 8: Проблемы с randomize.

```
1 : var
2 :   i:integer;
3 : begin
4 :   randomize;
5 :   for i:=1 to 5 do
6 :     writeln(random(50));
7 :   writeln;
8 :   randomize;
9 :   for i:=1 to 5 do
10:     writeln(random(50));
11:   writeln;
12: end.
```

Запустив этот пример, вы увидите, что две последовательности по 5 чисел полностью совпадают, т.е. процедура randomize после второго вызова сгенерировала то же самое число, что и после первого вызова. По идее randomize должна обращаться к системному таймеру и, следовательно, должны быть сгенерированы различные числа; но этого, как мы видим, не происходит. По всей видимости, просто процедура randomize работает неправильно.

Совет: не используйте randomize по несколько раз в одной и той же программе (например, не стоит эту процедуру вызывать внутри других подпрограмм).

5.5. Директивы компилятора

Директивы компилятора – это специальные выражения, позволяющие влиять на генерацию компилятором машинного кода. Например, можно менять размер стека, который выделяется программе во время выполнения, отключить проверку переполняемости стека и т.д.

Чтобы объявить директиву компилятору, надо написать {$Dir}, где вместо Dir надо поставить название директивы. Например, чтобы подключить возможность обработки вещественных чисел типа extended, надо указать компилятору в начале программы директиву {$N+}.

Некоторые директивы компилятор использует по умолчанию. Для того, чтобы узнать их, можно набрать комбинацию клавиш Ctrl+O+O.

5.6. Что мы уже знаем о структурном программировании

Вы знаете, что программа (на машинном языке) – это последовательность элементарных команд. Давайте теперь проанализируем структуру программного кода, написанного на ТР (все то же самое может быть перенесено на другие процедурные языки).

Во-первых, раздел описаний в программе на машинном языке отсутствует. Давайте теперь разбираться с исполняемым разделом. Исполняемый раздел – последовательность операторов. В принципе, простой оператор можно принять за высокоуровневую команду, поэтому если бы не было составных операторов, то исполняемая часть программного кода была бы подобна программе на машинном языке.

Настоящих нововведения в процедурных языках два:

1. Типизированность
2. Наличие структурных операторов

Структурные операторы разделяют исполняемый код на уровни, причем из внутреннего оператора можно переходить только к тому, в котором он непосредственно находится (например, из нельзя из цикла третьей степени вложенности перейти сразу к первому циклу).

Сразу напрашивается вопрос, как же реализуются повторяющиеся действия на ассемблере (или машинном языке). Оказывается, для этого есть специальная команда, которая позволяет перескочить от одной команды к другой без проверки каких бы то ни было условий. В ТР есть специальный оператор goto, выполняющий те же действия.

Мы посмотрим, как работает goto на примере вычисления факториала числа.

В разделе Label объявляются метки. Сами по себе они ничего не делают. Они нужны только для того, чтобы использовать оператор goto. Если выполнение программы дойдет до строки 13, то оператор goto M просто говорит ПК, что следующим оператором будет первый оператор после метки M, т.е. f:=f*i.

Пример 9: Использование goto.

```
1 : label
2 :   M;
3 : var
4 :   i,n,f:longint;
5 : begin
6 :   readln(n);
7 :   f:=1;
8 :   i:=1;
9 : M:
10:   f:=f*i;
11:   i:=i+1;
12:   if (i<=n) then
13:     goto M;
14:   writeln('f= ',f)
15: end.
```

Т.е. с помощью оператора goto мы можем организовать цикл. Ясно, что при программировании с использованием меток о разделенности уровней говорить не

приходится: с помощью goto можно выскочить из цикла любой степени вложенности. А в ассемблерах метки – необходимость.

Использовать оператор goto не рекомендуется, т.к. любая программа может быть написана без него, а он – лишь запутывает программный код.

Почему оператор goto вообще оставили в TP?

- Я думаю, что в то время, когда создавался Паскаль (70-е годы) еще много программистов писало программы с использованием goto, и Никлаус Вирт (разработчик Pascal) не рискнул убрать этот оператор. А в дальнейших версиях Pascal goto оставляли для совместимости с предыдущими версиями.

Низкоуровневой замены условному оператору в TP нет, хотя команда сравнения на ассемблере, разумеется, есть.

Думаю, что сейчас, после того, как вы изучили 1-й отдел книги, было бы полезно еще раз заглянуть в 0-ю главу и посмотреть уже опытным глазом на коды программ на разных языках программирования. Думаю, что вы сможете легко отыскать в ассемблерной программе и аналог goto, а может быть, и аналог if.

Глава 6: Массивы

С помощью циклов мы обрабатывали последовательности однотипных данных, но при этом мы не могли эту информацию сохранять; с помощью массивов можно осуществить нашу мечту: мы сможем хранить данные о котировках акций, дневные температуры за последние 100 лет, моделировать многочлены, быстро искать информацию и т.д.

6.1. Что такое массив?

- Массив – набор последовательно идущих переменных одного типа.

В TP массив описывается так:

имя типа = array [диапазон значений] of тип;

тип – любой тип TP (возможно, другие массивы).

В качестве индексных типов можно использовать любые порядковые типы, кроме longint и диапазонов с базовым типом longint.

Примеры:

```
type
  Mas= array [1..6] of real;
  Mas2= array ['a'..'s'] of integer;
var
  a,b:Mas;
```

В памяти массив A типа Mas представлен следующим образом:

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]

Однородность массива позволяет получить доступ к любому его элементу зная лишь тип переменных и адрес начала массива. Роль адреса начала массива играет его идентификатор (например, в предыдущем примере A указывает на начало массива[9]). Прибавляя смещение, умноженное на размер одной переменной, мы получим как раз адрес нужного элемента.

Массивы статичны, т.е. их размер не может изменяться во время работы программы. Это следует из того, что переменные размещаются в памяти компилятором и не могут быть затем удалены из программы. Поэтому если во время работы программы могут понадобиться массивы разных размеров, то надо объявлять массив наибольшего размера, либо использовать динамические структуры данных, которые мы изучим в главе 13.

Можно объявлять массивы и без непосредственного указания типа:

```
var
  a,b:array [1..6] of real;
```

Ограничения:

- Диапазон значений не может выходить за границы типа word.

Например:

```
Arr=array[0..6000] of integer;          {допустимое объявление}
ArrErr=array[70000..80000] of integer;  {недопустимое объявление}
```

[9] С точки зрения компилятора название переменной – это адрес первого байта, занимаемого ею в памяти компьютера.

- Общий размер массива, как и любой другой переменной не должен превышать 64 Кб – 16 байт (65520 байт)[10]

Например, размер массива arrErr=array[10000..60000] of integer; составляет 100000 байт, и его использовать нельзя.

Массивы одного типа можно присваивать один другому.

В качестве индексов можно использовать и целые числа, и символы (символьный тип будет изучаться позднее). Но начинать индексацию с -4, а тем более с ‘a’ в большинстве случаев крайне неудобно, поэтому мы будем работать с вами преимущественно с индексацией [1..n].

6.2. Примеры работы с массивами

Пример 1: Ввод массива с клавиатуры, печать на экран и нахождение суммы его элементов.

```
1 : const n=5; {n - количество элементов массива}
2 : type Mas=array[1..n] of integer; {Объявление нового типа}
3 : var
4 :   i:byte;
5 :   A:Mas; {Объявление переменной типа Mas}
6 :   S:integer;
7 : BEGIN
8 :   writeln('Введите, пожалуйста, ',n,' элементов массива');
9 :   for i:=1 to n do {В этом цикле user вводит элементы массива}
10:     read(A[i]);
11:   writeln('А вот и наш массив:');
12:   for i:=1 to n do  {Печать массива A на экран}
13:     write(A[i],' ');
14:   writeln;
15:   for i:=1 to n do  {Этот цикл находит сумму элементов массива}
16:     S:=S+A[i];
17:   writeln('Сумма элементов массива равна ',S);
18: END.
```

Вы видите, что размер массива объявлен константой n. Такой подход позволяет убить сразу двух зайцев: если мы захотим изменить размер массива, то можем поменять лишь значение константы n (если бы мы ее не использовали, то мы должны были бы пройтись по всему коду, и везде заменить одни числа на другие). Кроме того, могут быть другие константы, используемые в программе, которые совпадают со значением размера вашего массива. Поэтому сам процесс замены одних чисел на другие может стать источником трудно находимых ошибок.

Read(A[i]) позволяет пользователю ввести i-й элемент массива A. Поэтому когда i=1 вы будете вводить 1-й элемент массива, на следующем шаге, когда i=2, - элемент A[2] и т.д. Аналогично происходит вывод массива на экран и вычисление суммы элементов массива.

Замечание: ввести элементы массива, написав read(A) нельзя, так как с помощью процедуры read можно вводить лишь переменные простых типов.

---

[10] Это ограничение возникает из того, что TP работает под управлением MS-DOS. В Delphi, которая использует функции 32-разрядных операционных систем Windows такого ограничения на размер массива нет.

Пример 2: Вы держите деньги в n банках. В массиве A хранятся размеры ваших вкладов, а в массиве B – процентные ставки банков. Вы должны подсчитать свой доход к концу месяца.

```
1 : const
2 :   n=5;
3 : type
4 :   Mas=array[1..n] of real;
5 : var
6 :   A:Mas; {суммы вкладов в банках}
7 :   B:Mas; {процентные ставки в банках}
8 :   i:integer;
9 :   s:real;
10: begin
11:   for i:=1 to n do
12:     begin
13:     writeln('Введите ваш вклад в ',i,'-м банке');
14:     read(A[i]);
15:     writeln('Введите процентную ставку в (% годовых)');
16:     read(B[i]);
17:     end;
18:   for i:=1 to n do
19:     S:=S+A[i]*B[i];
20:   S:=S/1200;
21:   writeln('В конце месяца вы заработаете ',S,' грн.');
22: end.
```

По большому счету дело сводится к подсчету скалярного произведения массивов A и B, только его надо еще разделить на 1200 (12 месяцев, 100 процентов).

Пример 3: Программа, которая меняет порядок следования элементов в массиве на обратный. Например, если исходный массив 1 2 3 4 5, то результатом будет массив 5 4 3 2 1.

```
1 : const n=4; {n - количество элементов массива}
2 : type Mas=array[1..n] of integer; {Объявление нового типа}
3 : var
4 :   i:byte;
5 :   A:Mas; {Объявление переменной типа Mas}
6 :   tmp:integer;
7 : BEGIN
8 :   writeln('Введите, пожалуйста, ',n,' элементов массива');
9 :   for i:=1 to n do {В этом цикле user вводит элементы массива}
10:     read(A[i]);
11:   writeln('Исходный массив');
12:   for i:=1 to n do  {Печать массива A на экран}
13:     write(A[i],' ');
14:   writeln;
15:   for i:=1 to (n div 2) do {1-й элемент меняется с n-тым}
16:     begin                  {2-й с n-1-ым, и.т.д.}
17:     tmp:=A[i];
18:     A[i]:=A[n-i+1];
```

```
19:     A[n-i+1]:=tmp;
20:     end;
21:   writeln('Новый массив');
22:   for i:=1 to n do  {Печать массива A на экран}
23:     write(A[i],' ');
24: END.
```

Смысл программы такой: мы должны поменять первый элемент с последним, второй с предпоследним, и т.д. Но обратите внимание, что цикл надо прокручивать только до середины массива (что произошло бы с массивом, если бы в строке 15 стояло вместо n div 2 просто n?)

6.3. Моделирование многочленов

$P_n[x] = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$ - общий вид многочлена n-й степени от 1-й переменной.

Очевидно, что многочлен можно представить в виде массива коэффициентов. Размерность массива будем считать достаточно большой, чтобы не было проблем с реализацией любых арифметических действий над многочленами. Причем удобнее начинать нумерацию с конца массива, а не с его начала.

Например, многочлен $2x^2 + 3x + 4$ будет представлен (если максимальная степень многочлена = 3) в виде массива 0 2 3 4 (если максимальная степень многочлена больше трех, то нулей перед двойкой будет больше).

Есть много стандартных полиномов, использующихся особенно часто, например, полиномы Чебышева, задающиеся рекуррентной формулой:

$P_0(x) = 1$, $P_1(x) = x$, $P_n(x) = 2xP_{n-1}(x) - P_{n-2}(x)$, $n > 1$.

Вычислять многочлен Чебышева n-й степени можно так:

Пусть у нас есть $A = P_{n-2}(x)$, $B = P_{n-1}(x)$.

Надо вычислить $C = P_n(x) = 2xP_{n-1}(x) - P_{n-2}(x) = 2xB - A$

Для этого надо:

$C := 2x \cdot B - A;$

$A := B;$

$B := C;$

Теперь $A = P_{n-1}(x)$ и $B = P_n(x)$, - значит можно вычислять следующий многочлен Чебышева.

Но решение можно улучшить: вычисление коэффициентов массива $P_n(x)$ и присваивание их в массив B проводить параллельно с копированием массива B в массив A. В таком случае вместо вспомогательного массива будет достаточно одной переменной. В примере 4 этот алгоритм реализован на TP.

На i-м витке цикла (20-я строка) вычисляется полином (n-i)-й степени. Перед вычислением в массиве A храниться полином (n-i-2)-й степени, а в массиве B - (n-i-1)-й степени. В конце витка цикла в массиве B будет храниться многочлен (n-i)-й степени, а в массиве A - многочлен (n-i-1)-й степени.

В строках 22-27 вычисляются все коэффициенты многочлена Чебышева, кроме самого младшего. С помощью строк 24 и 26 в массив A присваивается массив B, а в строке 25 – вычисляются новые коэффициенты массива B.

Вычисление младших коэффициентов не укладывается в общую схему, поэтому их надо вычислять отдельно (в строках 28-30).

Пример 4: Вычисление многочлена Чебышева n-й степени.

```
1 : uses Crt;
2 : const  n=20;
3 : type
4 :   Poly=array [1..n] of integer;{Polynom - многочлен}
5 : var
6 :   p:integer;      {Potenz - степень}
7 :   tmp,i,j:integer;
8 :   A,B:Poly;
9 : begin
10:   Clrscr;
11:   writeln('Введите порядок многочлена Чебышева');
12:   readln(p);
13:   if (p=0) then {P(0)=1}
14:     begin
15:     writeln(1);
16:     exit;
17:     end;
18:   A[n]:=1;   {В массиве A хранится P(0)}
19:   B[n-1]:=1; {В массиве B хранится P(1)}
20:   for i:=n-2 downto n-p-1 do
21:     begin
22:     for j:=i to n do
23:       begin
24:       tmp:=B[j-1]; {сохраняем значение B[j-1]}
25:       B[j-1]:=2*B[j]-A[j-1];{используем рекуррентную формулу}
26:       A[j-1]:=tmp;
27:       end;
28:     tmp:=B[n]; {вычисляем последний коэффициент}
29:     B[n]:=-A[n];
30:     A[n]:=tmp;
31:     end;
32: {Печать многочлена}
33:   for i:=n-p to n do
34:     write(A[i], ' ');
35:   readln;
36: end.
```

Замечание: На самом деле вычислять полиномы Чебышева можно вообще с помощью лишь одного массива (см. упражнение 9).

Давайте теперь научимся производить арифметические действия с многочленами. Сейчас мы напишем программу, позволяющую умножать многочлены.

Пусть $A = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$, а $B = b_n x^n + b_{n-1} x^{n-1} + \ldots + b_1 x + b_0$. Произведение i-й компоненты многочлена A на j-ю компоненту многочлена B будет иметь вид $a_i b_j x^{i+j}$, поэтому нам надо лишь вычислить $a_i b_j$ и прибавить его к нужному элементу массива C. Главное – не запутаться с коэффициентами. На i-м месте массива стоит коэффициент при $n-i$-й степени многочлена. Поэтому $2n-i-j$ - это степень многочлена, при

котором должен стоять коэффициент $a_i b_j$, а $n-(2n-i-j)$ - индекс элемента в массиве С, к которому это $a_i b_j$ надо прибавить.

Пример 5: Умножение 2-х многочленов.

```
1 : uses Crt;
2 : const n=20;
3 : type
4 :   Poly=array [1..n] of integer; {Polynom - многочлен}
5 : var
6 :   p1,p2:integer; {Potenz - степень}
7 :   i,j:integer;
8 :   A,B,C:Poly;
9 : begin
10:   Clrscr;
11:   writeln('Введите степень первого многочлена');
12:   readln(p1);
13:   writeln('Введите коэффициенты первого многочлена');
14:   for i:=n downto n-p1 do
15:     readln(A[i]);
16:   writeln('Введите степень второго многочлена');
17:   readln(p2);
18:   writeln('Введите коэффициенты второго многочлена');
19:   for i:=n downto n-p2 do
20:     readln(B[i]);
21: {Теперь само умножение C:=A*B}
22:   for i:=n downto n-p1 do
23:     for j:=n downto n-p2 do
24:       begin
25:       C[n-(2*n-i-j)]:=C[n-(2*n-i-j)]+A[i]*B[j];
26:       end;
27: {Печать многочлена C}
28:   for i:=n-(p1+p2) to n do
29:     write(C[i], ' ');
30:   readln;
31: end.
```

Аналогично можно реализовать сложение, вычитание, деление, возведение в степень и другие операции над многочленами. Это будет одно из упражнений в главе «Модули и записи».

Я надеюсь, что вы ощутили мощь массивов и то, что ваши программистские мышцы растут не по дням, а по часам. Но, к сожалению, метод моделирования многочленов с помощью массивов не всегда хорош. Есть 2 проблемы:

1. Чтобы хранить многочлен $x^{1000}+x^{495}$ понадобится массив из 1001 элемента, а хотелось бы ограничиться 2-мя переменными, иначе вычисления будут идти значительно дольше, и расход памяти будет очень большим.
2. Если обобщать задачу, рассматривая многочлены не от одной, а от нескольких переменных (число которых также может быть любым, заранее не фиксированным числом), то массивы принципиально не подходят.
3. Всегда может возникнуть переполнение, если степень многочлена превосходит размер массива.

Все три проблемы можно решить, если использовать динамические структуры данных, например, списки (структура данных называется динамической, если ее размер может меняться в процессе работы программы). Но с ними мы познакомимся позже, а пока что будем довольствоваться тем, что есть.

Рассмотрим еще одну задачу, которая является подзадачей для «быстрой» сортировки массива, которую мы изучим в главе 8.

Пример 6: Дан целочисленный массив A. Надо переформировать его элементы так, чтобы вначале стояли положительные числа и нули, а затем шли отрицательные числа.

```
1 : const
2 :   n=10;
3 : type
4 :   Mas=array[1..n] of integer;
5 : var
6 :   i,j,tmp:integer;
7 :   A:Mas;
8 : begin
9 :   randomize;
10:   for i:=1 to n do
11:     A[i]:=random(n)-(n div 2);
12:   for i:=1 to n do
13:     write(A[i], ' ');
14:   writeln;
15:   i:=1;
16:   j:=n;
17:   while (i<j) do
18:     begin
19:     while (i<j) and (A[i]>=0) do {Ищем отрицательный элемент}
20:       inc(i);
21:     while (i<j) and (A[j]<0) do  {Ищем положительный элемент}
22:       dec(j);
23:     if (i>=j) then{если положительный элемент стоит дальше, }
24:       break;      {чем отрицательный, то можно выходить}
25:     tmp:=A[i];  {Меняем A[i] и A[j] местами}
26:     A[i]:=A[j];
27:     A[j]:=tmp;
28:     inc(i);   {сдвигаемся на соседние элементы}
29:     dec(j);   {без сдвигов ошибки тоже не будет}
30:     end;
31:
32:   for i:=1 to n do
33:     write(A[i], ' ');
34:   readln;
35: end.
```

Алгоритм прост: движеемся с обоих концов и ищем элементы, которые нарушают порядок, установленный по условию задачи. Как только такие числа найдены, меняем их местами и ищем следующую пару. Когда станет $i \geq j$ (т. е. индексы

встретятся или заползут друг за друга), то необходимый порядок следования будет достигнут. Детали реализации, я думаю, будут понятны из комментариев в программе.

В строках 28-29, после обмена местами найденных элементов, индексы сдвигаются на соседние элементы. Эти две строки не обязательны, - и без них программа будет работать правильно, но с ними – красивее.

6.4. Поиск в массиве

Найти элемент в массиве означает указать его индекс, или дать понять, что такого элемента нет. Индексация у массивов будет [1..n], поэтому если элемента в массиве нет, то будем считать, что его индекс = 0.

Допустим, задан массив целых чисел A и число x, которое надо найти. Наиболее естественный способ поиска заданного числа в массиве, - просто сравнивать поочередно каждый элемент $A[i]$ с числом x. Этот метод называется последовательным поиском.

Пример 7: Последовательный поиск.

```
1 : const n=4; {n - количество элементов массива}
2 : type Mas=array[1..n] of integer; {Объявление нового типа}
3 : var
4 :   i:byte;
5 :   A:Mas;
6 :   x:integer;
7 : BEGIN
8 :   writeln('Введите, пожалуйста, ',n,' элементов массива');
9 :   for i:=1 to n do {В этом цикле user вводит элементы массива}
10:     read(A[i]);
11:   writeln('Исходный массив');
12:   for i:=1 to n do  {Печать массива A на экран}
13:     write(A[i],' ');
14:   writeln;
15:   writeln('Введите число, индекс которого надо найти');
16:   readln(x);
17:   for i:=1 to n do
18:     if (A[i]=x) then
19:       begin
20:       writeln('у элемента ',x,' индекс =',i);
21:       exit;
22:       end;
23:   writeln('Элемента в массиве нет');
24:   readln;
25: END.
```

Очевидно, что последовательный поиск – единственно возможный способ отыскать элемент в массиве, о внутренней структуре которого ничего не известно. Удивительно, но реализация поиска, которая приведена в примере 7 – не самая эффективная. Улучшить скорость быстрого поиска вам предлагается в упражнении 8.

А вот упорядоченность массива (по неубыванию или невозрастанию) позволяет сократить время поиска во много раз. Итак, встречайте, - бинарный поиск.

Пусть для определенности массив будет отсортирован по неубыванию.

Идея метода такая: сначала сравниваем число x с центральным элементом массива A ($A_{центр}$). Если $x = A_{центр}$, то мы нашли индекс числа x. Если $x > A_{центр}$, то ясно, что элемент x, если он в массиве вообще имеется, может лежать только во 2-й половине массива, а если $x < A_{центр}$, то x может лежать лишь в первой части массива. Выходит, что за один этап алгоритма область отбора сокращена вдвое (отсюда и название – бинарный). Теперь, когда выбрана нужная половина массива, сравниваем x с центральным элементом этой половины, и т.д.

В итоге величина области поиска станет равна 1 элементу, и будет ясно, есть ли элемент x в массиве, и если да, то каков его индекс.

Пример 8: Бинарный поиск.

```
1 : const
2 :   n=34;
3 : var
4 :   M:array [1..n] of integer;
5 :   i,s:byte;
6 :   x:integer;
7 :   r,l,h:integer;
8 :   zahl:integer;  {Zahl - число}
9 : BEGIN
10:   {Заполнение случайными элементами по неубыванию}
11:   randomize;
12:   M[1]:=1;
13:   for i:=2 to n do
14:     M[i]:=random(10)+M[i-1];
15:
16:   writeln(' Исходный массив');
17:   for i:=1 to n do
18:     write(M[i],' ');
19:   writeln;
20:
21:   write('Введите число, которое надо найти');
22:   readln(zahl);
23:   {---------- Сам бинарный поиск ------------}
24:   l:=1;{Левая граница области поиска}
25:   r:=n;{Правая граница области поиска}
26:   while (l<=r) do{Пока границы не пересеклись}
27:     begin
28:     h:=(r+l) div 2;
29:     if (M[h]=zahl) then
30:       begin
31:       writeln('Индекс числа ',zahl,' = ',h);
32:       readln;
33:       exit;
34:       end;
35:     if M[h]>zahl then
36:       r:=h-1 {выбираем левую половину области поиска}
37:     else
38:       l:=h+1;{выбираем левую половину области поиска}
```

```
39:      end;
40:    writeln('Числа ', zahl, ' в массиве нет');
41:    readln;
42: end.
```

6.5. Сортировка массива

Т.к. упорядоченность массива значительно ускоряет поиск элементов, то сортировка массива становится первоочередной проблемой. Для того чтобы сортировать массивы есть много методов, различающихся по скорости работы.

Простейшие алгоритмы (пузырьковая сортировка, сортировка выбором и сортировка вставками) работают за время cn^2, где n - длина массива, а c - некоторая константа. Они очень медленны при больших n, но для маленьких массивов они работают достаточно быстро.

Для сортировок больших массивов применяются более быстрые сортировки (например, сортировка слиянием (см. упражнение 23 этой главы)), работающие за $cn\log_2 n$ действий. Очень хорошие результаты показывает быстрая сортировка (см. раздел 8.9.), которая в худшем случае требует cn^2 действий, но в среднем работающая очень быстро.

В некоторых случая можно применять сортировки, упорядочивающие массив за cn операций. С одной из них – сортировкой подсчетом – мы познакомимся в разделе 10.6.

В особых случаях можно применять комбинированные сортировки.

В этой главе мы рассмотрим 3 простейшие сортировки:

1. Сортировка методом пузырька (обмена)
2. Сортировка методом выбора
3. Сортировка методом вставок

У всех трех алгоритмов есть одно сходство: массив делится на 2 части – отсортированную и неотсортированную, причем с каждым шагом внешнего цикла количество отсортированных элементов увеличивается на 1.

В дальнейшем для определенности будем считать, что массив надо упорядочить по возрастанию.

6.6. Сортировка методом пузырька (обмена)

Алгоритм пузырька основан на следующем наблюдении: если два элемента стоят не в нужном порядке, то если их поменять местами, то массив станет более упорядоченным, чем прежде. Стало быть, чтобы отсортировать массив, надо только определить порядок, в котором будут обмениваться местами элементы. Самый простой алгоритм такой: сравниваем первый элемент со вторым, потом – второй с третьим и т.д. до конца массива. В результате, кроме того, что упорядоченность массива возрастет, можно с уверенностью сказать, что наибольший элемент точно будет находиться в конце массива (т.е. на своем месте). Поэтому сортировку и назвали пузырьковой: каждый раз по крайней мере один элемент всплывает на положенное место. Теперь можно провести еще одну точно такую же серию обменов, только пробегать можно до предпоследнего элемента, т.к. последний уже находится на своем месте.

Давайте посмотрим, как работает алгоритм на следующем массиве:

2	1	4	3	-2

1 этап (в примере j = n-1):

M[1] > M[2] => меняем элементы M[1] и M[2] местами:

1	2	4	3	-2

M[2] < M[3] => ничего не делаем:

1	2	4	3	-2

M[3] > M[4] => меняем элементы M[3] и M[4] местами:

1	2	3	4	-2

M[4] > M[5] => меняем элементы M[4] и M[5] местами:

1	2	3	-2	4

На этом первый этап (один виток внешнего цикла) заканчивается. После этого j становится равным n-2, и начинается второй этап. После окончания 2-го этапа массив примет вид:

1	2	-2	3	4

Думаю, что дальше алгоритм можно не расписывать. Ясно, что нужны еще 2 итерации алгоритма, чтобы массив был полностью отсортирован.

Пример 9: Сортировка методом пузырька.

```
1 : uses Crt;
2 : const
3 :   n=100;
4 : var
5 :   M:array [1..n] of integer;
6 :   i,j:word;
7 :   tmp:integer; {Вспомогательная переменная}
8 : BEGIN
9 :   Clrscr;
10:   randomize;
11:   for i:=1 to n do  {Заполняем массив случайными числами}
12:     M[i]:= random(100);
13:   writeln('Исходный массив:');
14:   for i:=1 to n do
15:     write(M[i],' ');
16:   writeln;
17: {Алгоритм сортировки методом пузырька}
18:   for j:=n-1 downto 1 do {j - граница области сортировки}
19:     for i:=1 to j do    {Проход по текущей области сортировки}
20:       if M[i]>M[i+1] then   {Если требуемый порядок нарушается,}
21:         begin               {то переставляем местами элементы}
22:         tmp:=M[i+1];
23:         M[i+1]:=M[i];
24:         M[i]:=tmp;
25:         end;
26:
27:   writeln('Отсортированный массив');
28:   for i:=1 to n do
29:     write(M[i],' ');
30:   readln;
31: end.
```

6.7.Сортировка выбором

Второй алгоритм еще проще, чем пузырек: на первом шаге ищется минимальный элемент в массиве, и меняется с первым элементом массива. После этого первый элемент уже находится на своем месте. Затем ищется элемент, который будет самым маленьким среди всех элементов массива, кроме первого. Этот элемент обменивается местами со вторым элементом массива и т.д.

Давайте испытаем сортировку выбором на примере:

2	1	4	3	-2

Этап 1: минимальный элемент – M[5]. Его надо поменять местами с M[1]. В результате получим массив:

-2	1	4	3	2

Этап 2: минимальный элемент (среди элементов M[2]…M[5]) – M[2]. Его надо поменять местами с самим собой:

-2	1	4	3	2

Этап 3: минимальный элемент (среди элементов M[3]…M[5]) – M[5]. Его надо поменять местами с M[3]. В результате получим массив:

-2	1	2	3	4

Массив уже отсортирован, но это только потому, что массив попался хороший. Так как мы рассматриваем общий случай, то надо провести еще одну итерацию алгоритма – поиск минимального элемента среди M[4],M[5].

Пример 10: Сортировка методом выбора.

```
1 : uses Crt;
2 : const
3 :   n=100;
4 : var
5 :   M:array [1..n] of integer;
6 :   i,j:word;
7 :   tmp:integer; {Вспомогательная переменная}
8 :   ind:word;
9 : BEGIN
10:   Clrscr;
11:   randomize;
12:   for i:=1 to n do  {Заполняем массив случайными числами}
13:     M[i]:= random(1000);
14:   writeln('Исходный массив:');
15:   for i:=1 to n do
16:     write(M[i],' ');
17:   writeln;
18: {Алгоритм сортировки методом выбора}
19:   for i:=1 to n-1 do {Внешний цикл изменяет область сортировки}
20:     begin
21:     ind:=i;   {индекс минимального элемента}
22:     for j:=i+1 to n do {Проходим по области поиска}
23:       if M[j]<M[ind] then {Если нашли элемент меньше M[ind]}
24:         ind:=j;  {Запоминаем индекс минимального элемента}
25:
```

```
26:     tmp:=M[ind]; {меняем местами минимальный элемент и m[i]}
27:     M[ind]:=M[i];
28:     M[i]:=tmp;
29:     end;
30:
31:   writeln('Отсортированный массив');
32:   for i:=1 to n do
33:     write(M[i],' ');
34:   readln;
35: end.
```

6.8. Сортировка вставками

Сортировка вставками напоминает расстановку карт по старшинству: берем подряд все карты, для каждой из них ищем подходящее место и вставляем ее на это место.

В сортировке вставками массив разбивается на 2 части: отсортированную (она будет находиться в начале массива) и неотсортированную (остальная часть массива). Сначала отсортированная часть массива – это первый элемент. Далее по очереди выбираются элементы из неотсортированной части, после чего для них ищется место в отсортированной части. Когда для элемента место найдено, его надо освободить. Для этого все элементы, начиная с этого индекса, надо сдвинуть вправо на 1 единицу. После этого элемент можно вставить на его законное место и перейти к следующему числу.

Давайте теперь отсортируем наш любимый массив вставками.

2	1	4	3	-2

Неотсортированная часть массива – это элементы M[2]…M[5].

1 этап: сохраняем число M[2] (нам оно понадобится) и ищем место для числа M[2] = 1. Он должен стоять первым. Теперь сдвигаем число M[1] на единицу вправо. Получим массив

2	2	4	3	-2

Осталось вставить число 1 (которое мы сохранили в дополнительной переменной) в первый элемент массива:

1	2	4	3	-2

2 этап: надо вставить в отсортированную часть элемент M[3] = 4. Конечно, мы с вами видим, что можно сразу перейти к следующему элементу, но компьютер то этого не знает, поэтому он добросовестно сохраняет M[3] в дополнительной переменной, потом ищет место в памяти, куда этот M[3] надо вставить, и т.д.

3 этап: надо найти подходящее место для элемента M[4]=3. Для этого надо сместить вправо M[3]:

1	2	4	4	-2

и вставить число 3 куда положено:

1	2	3	4	-2

Последнюю итерацию описывать не будем.

Пример 11: Сортировка методом вставок.

```
1 : uses Crt;
2 : const
3 :   n=100;
```

```
4 : var
5 :   M:array [1..n] of integer;
6 :   i,j,k:word;
7 :   tmp:integer; {Вспомогательная переменная}
8 : BEGIN
9 :   Clrscr;
10:   randomize;
11:   for i:=1 to n do  {Заполняем массив случайными числами}
12:     M[i]:= random(1000);
13:   writeln('Исходный массив:');
14:   for i:=1 to n do
15:     write(M[i],' ');
16:   writeln;
17: {Алгоритм сортировки методом вставок}
18:   for i:=2 to n do
19:     begin
20:     tmp:=M[i]; {запоминаем M[i]}
21:     j:=1;
22:     while tmp>M[j] do {ищем место, куда надо вставить M[i]}
23:       j:=J+1;
24:     for k:=i-1 downto j do {Сдвигаем элементы массива}
25:       M[k+1]:=M[k];       {чтобы освободить место для M[i]}
26:     M[j]:=tmp;   {Вставляем элемент на нужную позицию}
27:     end;
28:
29:   writeln('Отсортированный массив');
30:   for i:=1 to n do
31:     write(M[i],' ');
32:   readln;
33: end.
```

6.9. Сравнение простейших алгоритмов сортировки

Интересно, какой из алгоритмов сортировки, рассмотренных нами, работает быстрее всех. Для сравнения скорости их работы можно воспользоваться экспериментальным методом: проверять скорость работы алгоритмов для массивов различной длины. Так мы и поступим, когда в 8-й главе будем изучать быструю сортировку, т.к. ее анализ довольно сложен.

Второй метод анализа алгоритмов – теоретический: подсчитать, как зависит количество действий, которые требуется выполнить для того, чтобы отсортировать массив. Анализ сложных алгоритмов сортировки требует бОльших знаний математики, чем те, которые у нас есть, но с теми простыми алгоритмами, которые мы изучили, мы сможем справиться.

Один этап сортировок пузырьком и обменом заключается в просмотре неотсортированной части массива. На каждой итерации проводится по крайней мере операция сравнения. Следовательно, на i-м шаге будет проведено $n-i$ операций. Общее количество операций будет равно $1+2+3+...+n-1=\frac{n(n-1)}{2}$.

Аналогичные оценки справедливы и для сортировки вставками: на каждом этапе надо просмотреть часть массива для того, чтобы найти позицию, куда надо вставить

элемент, и сдвинуть оставшуюся часть массива. Значит, количество действий на i-м этапе будет равно i. Общее количество операций будет тоже $\frac{n(n-1)}{2}$.

В выкладках мы складывали операции сравнения с операциями пересылки данных (если бы использовались и арифметические операции, мы бы и их свалили в одну кучу). Это вполне правомерно, т.к. любая операция может быть представлена в виде конечного числа «универсальных операций» процессора. Поэтому в будущем если не будет указываться, какие именно операции имеются в виду, то это – универсальные операции. Конечно, при более детальном анализе полезно знать отдельно количество сложений, умножений и т.д.

Итак, время работы всех сортировок равно $c\frac{n(n-1)}{2}$. При больших n $\frac{n(n-1)}{2} \approx \frac{n^2}{2}$, поэтому приближенное время работы будет $c_1 n^2$ (константа c_1 зависит от алгоритма сортировки).

Мы доказали, что время сортировки массива для трех рассмотренных алгоритмов пропорционально квадрату его размера. Но мы еще не знаем, какой из них работает быстрее. Сейчас мы с вами докажем, что сортировка вставками работает быстрее, чем сортировка пузырем.

- Назовем транспозицией перестановку соседних элементов.

В качестве меры неупорядоченности массива выберем минимальное количество транспозиций, которое необходимо для того, чтобы упорядочить массив.
Например: чтобы упорядочить массив
1 4 3 2 8
достаточно сделать 3 транспозиции.

Теперь можно перейти к анализу алгоритмов. Пусть дан массив размера n степени неупорядоченности x.

Чтобы его отсортировать методом пузырька, надо сделать $\frac{n(n-1)}{2}$ сравнений и x перестановок ($3x$ присваиваний).

Для сортировки вставками надо $\frac{n(n-1)}{2}-x$ сравнений и $x+2(n-1)$ присваиваний.

Уже сейчас можно сказать, что сортировка вставками значительно лучше, чем сортировка пузырьком, т.к. при больших x количество присваиваний и сравнений значительно меньше.

Анализировать сортировку выбором значительно труднее, так как два массива одинакового размера с одинаковой степенью неупорядоченности могут быть отсортированы за разное количество действий. Например, массивы
5 2 3 4 1 и 5 2 4 1 3
можно отсортировать за 7 транспозиций, но первый массив сортировка выбором упорядочит быстрее. Эти и другие сложности затрудняют анализ, поэтому мы не будем на нем останавливаться (есть гораздо лучшие алгоритмы сортировки).

Задачи

1. Заполнить массив А числами 1, 2, 3, …, n, где n – длина массива.
2. Дан массив A[n]. Найти разность между максимальным и минимальным элементами массива.
3. Найти среднее арифметическое и среднее геометрическое элементов массива.
4. Дан массив A[n], упорядоченный по неубыванию. Найти количество различных чисел среди элементов этого массива.
5. Найти сумму чисел, порядковый номер которых – число Фибоначчи.
6. Если вы запустите программу вычисления многочленов Чебышева для достаточно больших n (например, 10…15), то вы увидите, что при четных n в многочлен входят только четные степени x, а при нечетных n - нечетные степени x. Докажите этот результат для любых чисел n (т.е. докажите, что $T_n(-x) = (-1)^n T_n(x)$).
7. **(!)** Докажите следующее предложение:
   - если $f(-x) = -f(x)$, то $\int_{-l}^{l} f(x)dx = 0$
   - Используя предыдущее утверждение и результат предыдущей задачи, докажите, что если $g(-x) = g(x)$, и $m+n$ нечетно, то $\int_{-1}^{1} g(x)T_m(x)T_n(x)dx = 0$.
8. (Модернизация последовательного поиска). На каждом проходе цикла неявно проводится сравнение, достиг ли счетчик конца массива, или нет. На эти сравнения тратится время. Этого можно избежать: если в последний элемент массива записать число x, то элемент в массиве точно будет найден, и бесконечного цикла не будет. Используя эту идею, модернизируйте алгоритм последовательного поиска.
9. Используя результат задачи №6 предложите способ вычисления коэффициентов многочлена Чебышева без использования дополнительных массивов, и реализуйте его на ТР.
10. Дан целочисленный массив (не отсортированный). Сосчитать и напечатать, сколько различных чисел в этом массиве. Число действий должно быть порядка m^2.
11. Дан массив целых чисел из m+n элементов, рассматриваемый как соединение двух его отрезков: начала A[1]..A[m] из m элементов и конца A[m+1]..A[m+n] из n элементов. Не использую дополнительных массивов, переставить начало и конец. Число действий должно быть порядка m+n.
12. Даны два массива длины m и n соответственно, отсортированные по неубыванию. Найти количество общих элементов в массивах.
13. В примерах на работу с многочленами полиномы печатались в виде массива коэффициентов. Напишите программу, которая бы по массиву коэффициентов печатала бы сам многочлен. Например: по массиву 1 0 -3 4 7 -3 0 надо напечатать x^6-3*x^4+4*x^3+7*x^2-3*x.
14. Некоторое число содержится в каждом из трех неубывающих массивов $x[1] \le x[2] \le ... \le x[n]$, $y[1] \le y[2] \le ... \le x[m]$, $z[1] \le z[2] \le ... \le z[p]$. Найти это число. Число действий должно быть порядка $n+m+p$.
15. Задан массив натуральных чисел P. Найти минимальное натуральное число, не представимое суммой никаких элементов массива P. В сумму элементы массива могут входить только по одному разу.

16. Задан числовой массив. Найти отрезок максимальной длины, в котором первое число равно последнему, второе – предпоследнему и т. д. Вывести длину отрезка.
17. Даны два упорядоченных по неубыванию массива длины m и n соответственно. Образовать новый массив из $m+n$ элементов, также упорядоченный по неубыванию. Алгоритм должен быть оптимальным.
18. Отсортировать массив с помощью метода пузырька с подсчетом перестановок на каждом витке цикла. Если на некотором шаге перестановок нет, то это означает, что массив уже отсортирован.
19. Представьте себе, что исходный массив имеет такой вид: 9 0 7 1 3 5 6 6 10 5. Хотелось бы не менять 9 со всеми элементами по очереди, пока не доберемся до 10, а сделать как в сортировке вставками: найти первый элемент, больший 9, и вставить его перед ним, просто сдвинув на 1 все числа, стоящие между ними. Скорость сортировки при этом возрастет. Если число промежуточных чисел равно x, то по стандартному алгоритму пузырька надо сделать x операций сравнения и 3*x операций присваивания, а по нашему хитрому улучшению только x сравнений и x операций присваивания (еще x увеличений на 1, т.к. надо искать индекс – но это не страшно).
20. Модернизируйте сортировку вставками: ищите позицию в отсортированной части массива не последовательным поиском (как в примере 11), а бинарным.
21. Отсортировать массив так, чтобы в начале массива стояли положительные элементы в порядке убывания, а в конце стояли неположительные элементы в порядке возрастания. Разработать оптимальный алгоритм.
22. Написать программу, которая по заданному натуральному числу будет возвращать число, у которого отсортированы все цифры (14423 –> 12344). Массивы в программе не использовать.
23. (**Сортировка слиянием**) Если разбить массив на 2 половины и отсортировать их отдельно, а затем применить алгоритм из задачи 17, то скорость сортировки возрастет. Используя это наблюдение, разработайте алгоритм, который будет сортировать массив за время не cn^2, как рассмотренные нами алгоритмы, а за $c_2 n \log n$.

Глава 7: Подпрограммы

Вы знаете, что подпрограмма – это фрагмент кода, который можно вызывать из основной программы (а также других подпрограмм).

Подпрограммы позволяют программисту мыслить не в терминах отдельных операторов, а в терминах блоков операторов, выполняющих определенные действия. Кроме того, использование подпрограмм позволяет создавать библиотеки подпрограмм, которые могут использоваться в дальнейшем для других проектов. Причем желательно, чтобы подпрограммы были как можно более общими (сортировать не массивы из 10 элементов, а массивы из любого числа элементов любого типа, причем в произвольном порядке).

7.1. Описание подпрограмм

Описание процедур

```
procedure Имя (Список формальных параметров);
label         {Описание меток}
const         {Описание констант}
type          {Описание типов}
var      {Описание переменных}
procedure     {Блок описания  внутренних процедур }
function      {и функций}
begin {Исполняемая часть}
{Операторы}
end;
```

Описание функций

```
function Имя (Список формальных параметров): тип результата;
label         {Описание меток}
const         {Описание констант}
type          {Описание типов}
var      {Описание переменных}
procedure     {Блок описания  внутренних процедур }
function      {и функций}
begin {Исполняемая часть}
{Операторы, один из которых присваивает имени функции значение
результата}
end;
```

Вы видите, что по синтаксису описания процедуры и функции напоминают синтаксис описания основной программы: в них тоже можно описывать переменные, константы, типы и даже другие подпрограммы.

- Переменные, которые описываются внутри подпрограмм, называются локальными переменными.
- Переменные, объявляемые в основной программе, называются глобальными переменными.

Глобальные переменные существуют на протяжении всего времени работы программы, а локальные создаются после вызова подпрограммы, а непосредственно

перед окончанием ее работы они стираются из памяти. Более подробно различие между локальными и глобальными переменными будет рассмотрено позднее.

Пример 1: Несколько простых процедур и функций.

```
1 : uses Crt;
2 :
3 : procedure Hallo;  {Hallo = Привет}
4 : begin
5 :   writeln('Привет!');
6 : end;
7 :
8 : function Inhalt(R:real):real;{Inhalt - площадь, объем}
9 : begin
10:   Inhalt:=Pi*R*R;
11: end;
12:
13: function Faktor(n:byte):longint; {вычисление факториала}
14: var
15:   i:byte;
16:   f:longint;
17: begin
18:   f:=1;
19:   for i:=2 to n do
20:     f:=f*i;
21:   Faktor:=f;
22: end;
23:
24: {Вычисляет периметр прямоугольного треугольника по 2-м катетам}
25: function Perimeter(a,b:real):real;
26: begin
27:   if (a<0) or (b<0) then
28:     begin
29:     writeln('Вы ввели неверные данные');
30:     halt;
31:     end;
32:   perimeter:=a+b+sqrt(a*a+b*b);
33: end;
34:
35: Begin
36:   Hallo;
37:   writeln('Площадь круга радиусом 5 ед. = ',Inhalt(5));
38:   writeln(Faktor(4));
39:   writeln(Perimeter(3,4));
40: END.
```

Эта программа состоит из одной процедуры, трех функций, и основного блока (между begin и end.)

Начинается выполнение программы, как всегда, с основного блока (строка 35). В нем мы можем вызывать написанные нами подпрограммы (сами по себе они не вызываются).

В строке 36 вызывается подпрограмма Hallo. Вызов подпрограммы производится так: управление передается в подпрограмму, выполняются все ее операторы, а затем управление передается обратно в основную программу. В нашем случае после вызова Hallo будет напечатано сообщение “Привет!“, а затем выполнение программы продолжится с оператора на строке 37.

Давайте теперь рассмотрим описание функции Inhalt.

После названия функции в скобках следует список формальных параметров.

В большинстве случаев в подпрограммы надо передавать дополнительные параметры, которые необходимы для их работы, например, в функцию sqrt надо передать значение числа, корень которого эта функция должна извлечь.

Параметров может быть несколько: если функция должна вычислять площадь прямоугольника по заданным длинам сторон, то параметров должно быть 2.

Передаются параметры в скобках после названия подпрограммы.

В функции Inhalt один вещественный параметр – радиус круга (R:real). Передавая его в функцию, можно получать площади кругов разных радиусов.

После списка формальных параметров должен стоять тип возвращаемого значения. В данном случае – тоже real.

Замечание: процедуры не возвращают значений, поэтому типа возвращаемого значения у них нет.

Функция в результате своей работы возвращает значение, которое находится в переменной с названием, совпадающим с названием функции.

В строке 10 в переменную Inhalt записывается πR^2. Именно это значение и будет результатом функции Inhalt.

Замечание: Не забывайте записывать результат работы функции перед выходом из нее в соответствующую переменную.

Передача параметров в функцию осуществляется следующим образом: внутри функции создается переменная R типа real (это – **формальный параметр**), затем в нее копируется то значение параметра, которое вы в эту функцию передали (это – **фактический параметр**). Например:

Inhalt(5)   В переменную R будет записано значение 5
Inhalt(x)   В переменную R будет записано значение переменной x.

Замечание: часто считают, что если формальный параметр - это переменная R – то и в функцию в качестве параметра могут передаваться лишь значение переменной R. Это неверно: безразлично от того, какую переменную или константу мы передадим в подпрограмму, все равно внутри нее будет создана переменная R, в которую будет скопировано значение фактического параметра, и затем работа внутри функции будет проводиться лишь с этой только-что-созданной переменной R, т.е. значения фактических параметров изменяться не будут.

В 37-й строке вызывается функция Inhalt, и результатом ее будет значение площади круга с радиусом 5.

В 38-й строке вызывается функция вычисления факториала. В результате этого управление переносится в функцию Faktor. При этом создается, как вы уже знаете, переменная n типа byte, и в нее записывается 4. Но, кроме нее, будут созданы переменные i и f, объявленные в функции (до вызова функции этих переменных не существует).

После того, как все необходимые приготовления выполнены, отрабатывает сама функция, и в строке 21 результат записывается в переменную Faktor. После окончания работы функции Faktor в место вызова функции (строку 38) возвращается значение переменной Faktor, а затем все переменные, созданные внутри функции Faktor, стираются из памяти.

В 39-й строке вызывается функция, вычисляющая периметр прямоугольного треугольника по двум 2 катетам (тип real). Однотипные параметры в TP можно перечислять через запятую (см. строку 25).

В этой функции проводится проверка правильности входных параметров (для наглядности она не была приведена предыдущих функциях). Если хотя бы один из катетов меньше 0, то выдается сообщение об ошибке, и затем вызывается процедура halt, осуществляющая выход из программы. Раньше мы с вами пользовались процедурой exit. Но дело в том, что exit выходит только из подпрограммы, а не из программы полностью, поэтому раньше, когда вся наша программа состояла из одного блока, функции halt и exit работали бы одинаково.

Если значения параметров допустимы, то в переменную perimeter записывается значение периметра прямоугольного треугольника, которое и возвращается функцией.

Давайте теперь подытожим все вышесказанное:

При вызове подпрограммы:
1. Выделяется память под формальные параметры и локальные данные.
2. Значения фактических параметров копируются в память, выделенную для формальных.

Во время работы программы:
1. Значения переменных фактических параметров не меняются, значения переменных формальных параметров – изменяются.

При окончании работы подпрограммы:
1. Память, выделенная под формальные параметры и локальные данные очищается.
2. Новые значения формальных параметров, полученные в процессе работы подпрограммы, теряются вместе с очисткой памяти.

Кроме статической памяти, в которой размещается программа и глобальные переменные, программе выделяются дополнительные области памяти – стек и куча (или динамическая память). Все локальные переменные размещаются в стеке, причем всю работу по их размещению и удалению выполняет компилятор (т.е. он генерирует соответствующий набор команд на языках низкого уровня). Как работать с кучей мы рассмотрим в главе 13.

7.2. Передача массивов в подпрограммы

В предыдущем разделе этой главы вы научились передавать в подпрограммы параметры. Но мы с вами в качестве параметров использовали только числа. В принципе, передача массивов способом, описанным в предыдущей главе, не запрещена. Но давайте теперь посмотрим, что произойдет при этом. Предположим, что описана следующая процедура (то, что она делает, для нас не важно):

```
Procedure Test(M:Mas);
```

Здесь M – некоторый массив. При вызове процедуры Test(A); будут выполнены следующие действия:

1. Будет выделено место в стеке под формальный параметр M
2. В массив M будут скопированы соответствующие значения массива A.
3. Будет выполнено тело процедуры
4. Массив M будет стерт из стека.

Проблемы, возникающие при таком подходе

1. Создание дополнительного массива сильно расходует память (особенно, если массив достаточно большой).
2. Увеличивается время работы программы (тратится время на копирование данных из одного массива в другой).

Кроме того, есть еще одна принципиальная проблема. Так как внутри процедуры мы работаем с формальными параметрами, а не фактическими, то выходит, что с помощью описанного выше подхода нельзя написать, например, процедуру заполнения массива.

Пример 2: Проблема заполнения массива числами (füllen - заполнять).

```
1 : uses Crt;
2 : const
3 :   n=5;
4 : type Mas=array[1..n] of integer;
5 :
6 : var
7 :   K:Mas;
8 :   i:byte;
9 :
10: procedure FulleMas(A:Mas);
11: var
12:   i:integer;
13: begin
14:   for i:=1 to n do
15:     A[i]:=i;
16:   writeln('Массив A (формальный параметр) после заполнения');
17:   for i:=1 to n do
18:     write(A[i],' ');
19:   writeln;
20: end;
21:
22: BEGIN
23:   Clrscr;
24:   writeln('Массив  K до вызова процедуры FulleMas');
25:   for i:=1 to n do
26:     write(K[i],' ');
27:   writeln;
28:
29:   FulleMas(K);
30:   writeln('Массив  K после вызова процедуры FulleMas');
31:   for i:=1 to n do
32:     write(K[i],' ');
33: End.
```

До вызова процедуры FulleMas(K) в 29-й строке массив K заполнен нулями. После вызова процедуры будет создан массив A, в который будут скопированы значения массива K, затем массив A будет заполнен числами от 1 до n (значения массива K при этом не изменяются). После окончания процедуры массив A исчезнет из памяти и его значения потеряются. В результате массив K не изменился, а процедура FulleMas не сделала ничего: все результаты ее работы пропали даром. Запустите программу и убедитесь, что все произойдет именно так, как я сказал.

Для того чтобы решить все вышеперечисленные проблемы, в TP есть возможность передавать параметры способами, отличными от того, которым мы пользовались до этого.

7.3. Параметры подпрограмм

По способу передачи в подпрограммы, параметры делятся на три типа: параметры-значения, параметры-переменные (Var-параметры), параметры-константы (Const-параметры). До этого мы рассматривали передачу параметров по значению. Теперь остановимся на двух других типах параметров.

Как вы знаете, массив – это последовательность переменных одного типа, последовательно расположенных в памяти ПК. При этом название массива – это ссылка на его начало, и с помощью смещения относительно начала массива, указываемого в скобках, можно получить доступ к любому его элементу.

Если бы мы могли передать в подпрограмму сам адрес начала массива, то мы могли бы с помощью смещения относительно этого адреса получать доступ ко всем элементам массива, который мы хотели передать в подпрограмму. Параметры, передаваемые в подпрограмму таким образом, называются параметрами-переменными.

Теперь давайте опишем принцип работы таких параметров на примере:

Пример 3: Основные процедуры и функции для работы с массивами.

```
1 : const
2 :   n=3;
3 : type
4 :   Mas=array[1..n] of integer;
5 :
6 : var
7 :   M:Mas;
8 :   q:integer;{Глобальная переменная}
9 :
10: procedure LeseMas(var A:Mas); {Ввод элементов массива}
11: var                          {lesen = читать}
12:   i:integer;
13: begin
14:   for i:=1 to n do
15:     read(A[i]);
16: end;
17:
18: {Заполнение массива случайными числами от 0 до k-1}
19: procedure RandFulle(var A:Mas;k:word);
20: var
21:   i:integer;
```

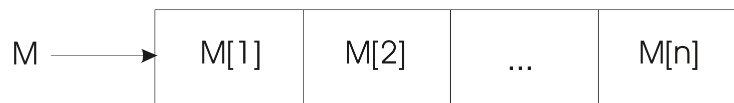
```
22: begin
23:   for i:=1 to n do    {Заполняем массив случайными числами}
24:     A[i]:=random(k);  {от 0 до k-1}
25: end;
26:
27: procedure SchrMas(const A:Mas);{Печать массива}
28: var                            {schreiben = писать}
29:   i:integer;
30: begin
31:   for i:=1 to n do
32:     write(A[i],' ');
33:   writeln;
34: end;
35:
36: {Вычисление НОД всех чисел массива}
37: function GGTMas(const A:Mas):integer;
38: {Вложенная функция вычисления НОД двух чисел}
39: function GGT(a,b:integer):integer;
40: var
41:   q:integer;{Эта переменная - локальная по отношению}
42:             {к функции GGTMas}
43: begin
44:   repeat
45:     q:=a mod b;
46:     a:=b;
47:     b:=q;
48:   until q=0;
49:   GGT:=a;
50: end;
51:
52: var {Переменные функции GGTMas}
53:   q:integer;{локальная переменная по отношению к}
54: {основной программе, и глобальная по отношению к}
55: {функции GGT}
56:   i:integer;
57: begin {Основной блок ф-ии GGTMas}
58:   q:=A[1];      {НОД 1-го числа есть само число}
59:   for i:=2 to n do  {Вычисляем НОД чисел массива}
60:     begin
61:     q:=GGT(q,A[i]);
62:     if q=1 then
63:       break;
64:     end;
65:   GGTMas:=q;
66: end;
67:
68: begin
69: {  writeln(GGT(34,48));}
70:   randomize;
71:   LeseMas(M); {ввод массива M}
72:   SchrMas(M);{печать массива M}
73:   q:=GGTMas(M);{ищем НОД элементов массива}
```

```
74:   writeln('НОД чисел массива - ',q);
75:   RandFulle(M,2900);{Заполняем массив M случайными}
76:   SchrMas(M);     {числами в диапазоне от 0 до 2899}
77:   q:=GGTMas(M);
78:   writeln('НОД чисел массива - ',q);
79: end.
```

В примере описаны простейшие подпрограммы, позволяющие удобно работать с массивами. Рассмотрим процедуру LeseMas (строка 10). Вы видите, что в процедуру передается один параметр – массив, который надо заполнить числами. Перед названием переменной стоит слово var. Это означает, что параметр будет передаваться как параметр-переменная. Если бы слово var не стояло, то мы бы передали бы массив M по значению, и результат был тот же, что и в примере №2 (данные потеряются).

Механизм работы var-параметров

Массив M хранится в памяти ПК следующим образом:



M (название массива) – это ссылка на начало массива M.

В строке 71 вызывается процедура LeseMas(M). Так как массив M передается в процедуру как параметр переменная, то в процедуре создается переменная A (формальный параметр), но не массив (как было бы, если бы параметр передавался по значению), а лишь ссылка на массив M, и в нее копируется адрес переменной M. Это можно показать на рисунке так:

Теперь можно обращаться к элементам массива M с помощью переменной A: A[i] – это элемент, находящийся в i-й позиции от адреса, записанного в переменной A, т.е. M[i]. Короче говоря, M[i] = A[i].

Внутри процедуры идет работа с переменной A, как с массивом, но фактически изменяться будут значения массива M. Когда процедура отработает, все переменные,

которые были созданы внутри процедуры, должны быть стерты из памяти. Ссылка A будет удалена, но все изменения, которые были с ее помощью сделаны с массивом M, сохранятся.

Итак, после передачи массива M в процедуру LeseMas будет создана ссылка A на начало массива M. Затем мы вводим с клавиатуры элементы A[i], которые, в то же время являются элементами массива M[i]. После окончания работы процедуры LeseMas ссылка A будет удалена, но введенные элементы в массиве M останутся.

Рассмотрим теперь процедуру RandFulle.

В процедуре два параметра. Обратите внимание на синтаксис заголовка: параметры различных типов должны разделяться точкой с запятой. Массив передается как var-параметр, а число k, – как параметр-значение.

В процедуру SchrMas массив передается как const-параметр.

Принцип работы параметров-констант тот же, что и параметров-переменных, единственное отличие, что с помощью const-параметров нельзя изменять значения объектов.

Переменные простых типов (числа, символы, булевские переменные) обычно передаются как параметры-значения.

Переменные структурированных типов (массивы, матрицы, записи) должны передаваться как параметры-переменные или как параметры-константы.

Считается, что передача параметров как параметров-переменных таит в себе опасность того, что данные, изменять которые внутри подпрограммы нельзя, все же будут каким-то образом подпорчены. При передаче параметров по значению такой проблемы нет. Именно поэтому, в целях защиты входных данных от «несанкционированного использования», были введены параметры-константы. Разумеется, при грамотном программировании никаких проблем при использовании var-параметров не возникнет.

Рассмотрим, наконец, функцию вычисления НОД всех элементов массива (GGTMas). Алгоритм вычисления основывается на формуле

$$НОД(x_1, x_2, ..., x_n) = НОД(НОД(x_1, x_2, ..., x_{n-1}), x_n)$$

Базовой операцией будет нахождение НОД 2-х чисел. Так как она представляет самостоятельный интерес, то имеет смысл написать отдельную функцию вычисления НОД 2-х чисел (НОД = GGT = der grosste gemeine Teiler). Эта функция – вложенная в функцию GGTMas, и поэтому может быть вызвана из только из функции GGTMas. При вызове функции GGT из основной программы будет выдана ошибка (чтобы убедиться в этом раскомментируйте строку 69) .

Разумеется, логичнее было бы сделать процедуру GGT внешней подпрограммой, т.к. вычисление НОД двух чисел возникает гораздо чаще, чем вычисление НОД трех и более чисел. Я сделал GGT вложенной функцией лишь для того, чтобы продемонстрировать вам, как можно объявлять вложенные подпрограммы.

Алгоритм работы GGTMas следующий: НОД одного числа – это само число. Поэтому в переменную q, в которой мы будем хранить значение НОД, сначала надо записать A[1]. Затем в цикле проходим по всем числам массива, начиная со 2-го. На i-м витке цикла в q будет храниться значение $НОД(A[1], A[2], ..., A[i-1])$. Строка 61 соответствует формуле $НОД(A[1], A[2], ..., A[n]) = НОД(НОД(A[1], A[2], ..., A[n-1]), A[n])$.

7.4. Локальные и глобальные переменные

В примере 3 объявлены 3 переменные q (строки 8, 41, 53). При этом они не конфликтуют между собой: все три занимают различные позиции в памяти, и компилятор всегда четко знает, какую из переменных и когда ему надо использовать.

- Пусть задана подпрограмма Unt. Подпрограмма, внутри которой описана Unt, называется глобальной по отношению к Unt. Переменные этой подпрограммы являются переменными, глобальными по отношению к Unt.
- Переменные и подпрограммы, объявленные внутри Unt называются локальными по отношению к Unt.

Налицо относительность понятий «локальный» и «глобальный». Например, переменные, объявленные внутри функции GGTMas будут локальными по отношению к основной программе, но в то же время глобальными по отношению к функции GGT. А все переменные внутри функции GGT будут локальными по отношению и к функции GGTMas и к основной программе.

Если встречаются глобальные и локальные переменные с одинаковыми именами, то использоваться будет переменная, объявленная в текущей процедуре. Если таковой нет, то будет использоваться переменная, объявленная в ближайшей из внешних процедур, если и там нет переменной с нужным названием, то поднимаемся еще на один уровень выше, и берем переменную оттуда, и т. д., так что при выборе переменной никакой неопределенности для компилятора не возникает.

Замечу, что все локальные переменные находятся в стеке – никаких подстеков для подподпрограмм нет.

7.5. Открытые параметры-массивы

Вы уже умеете передавать в процедуры массивы. Но, задавая тип массива, вы автоматически определяете его длину. Это очень серьезное ограничение, т.к. в одной программе может использоваться несколько массивов с элементами одного и того же типа данных но с разной длиной. Если действовать описанными выше методами, то надо объявить несколько различных типов массивов, и для каждого из них писать отдельные процедуры ввода/вывода элементов, их вывода и т. д. В TP необходимости писать один и тот же код для разных типов массивов можно избежать, если использовать открытые параметры-массивы.

При передаче параметров как открытых массивов в подпрограмму передается сам массив (ссылка на его начало), а также количество элементов в нем. Количество элементов передается неявно (т.е. в коде этот параметр не виден), но его можно узнать, используя функцию High.

Рассмотрим простой пример:

Пример 4: Использование открытых параметров-массивов.

```
1 : const
2 :   n=6;
3 :   m=12;
4 : type
5 :   Mas1=array [1..n] of real;
6 :   Mas2=array [1..m] of real;
7 : var
```

```
8 :   A:Mas1;
9 :   B:Mas2;
10: procedure SchrMas(const v:array of real);
11: var
12:   i:integer;
13: begin
14:    for i:=0 to High(v) do
15:      writeln(v[i]:10:4);
16: end;
17:
18: procedure EinfFulle(var v:array of real);{einfach - простой}
19: var
20:   i:integer;
21: begin
22:   for i:=0 to High(V) do
23:     v[i]:=i+1;
24: end;
25:
26: begin
27:   EinfFulle(A);
28:   SchrMas(A);
29:   writeln;
30:   EinfFulle(B);
31:   SchrMas(B);
32: end.
```

Посмотрите на заголовки обеих процедур примера (var v:array of real). Указан лишь тип элементов массива, а сам тип массива, передаваемого в процедуру, может быть произвольный.

Достигается общность следующим образом: минимальный индекс массива, передаваемого как открытый, всегда полагается равным 0, а индекс последнего элемента массива можно вычислить с помощью функции High(A), если A – это некоторый массив с типом элементов, указанным в заголовке (в данном примере – real). EinfFulle (einfach – простой) заполняет элементы числами 1, 2, …, High(A)+1.

Замечание: хотя длина передаваемого массива может быть любой, но тип элементов должен совпадать с типом, указанным в заголовке процедуры.

Преимущества открытых массивов особенно важны при написании модулей, которыми мы займемся через одну главу.

7.6. Совместимость и приведение типов

- Два типа называются совместимыми, если выполняется как минимум одно из условий:
  - оба типа одинаковые
  - оба типа вещественные
  - оба типа целочисленные
  - один из типов является поддиапазоном другого
  - оба типа – отрезки одного и того же базового типа.
  - оба типа – множественные типы с совместимыми базовыми типами

- один тип – Pointer, другой – любой ссылочный тип.

(Что такое Pointer и множественный тип вы еще не знаете, - эти типы данных будут рассмотрены в следующих главах).

Совместимость типов проверяется, например, при передаче параметров в подпрограммы. Если вы передадите фактический параметр с типом, не совместимым с типом формального параметра, то будет выдана ошибка Type mismatch.

По большому счету, совместимость по типу говорит об сходстве внутренней структуры типа. Например, целое число и вещественное число имеют с точки зрения организации данных совершенно различную структуру, поэтому операции умножения и сложения для целых и вещественных чисел различны.

Разумеется, надо проверять совместимость типов и при операции присваивания. Но в этом случае мы сталкиваемся с серьезными ограничениями: мы не можем, скажем, присвоить вещественному числу целое, что логично с точки зрения компьютера, но при этом очень неудобно. Поэтому вводится другое понятие – совместимость по присваиванию.

- Типы T1 и T2 называются совместимыми по присваиванию, т. е. можно выполнить операцию V1:=V2, где V1 – переменная типа T1, V2 – переменная типа T2.

Типы T1 и T2 будут совместимыми по присваиванию, если выполняется одно из следующих условий:

- T1 = T2 , и при этом T1 – не файловый тип.
- T1 и T2 – вещественные, и значения T2 входят в диапазон значений T1.
- T1 и T2 – целые, и значения T2 входят в диапазон значений T1.
- T1 – вещественный, T2 – целочисленный.
- T1 и T2 – строковые.
- T1 и T2 – совместимые типы указателей.
- T1 и T2 – совместимые множественные типы, и T2 – подмножество T1.
- T1 – строковый, T2 – char.

Правда, довольно часто приходится присваивать переменные несовместимых по присваиванию типов. Для таких случаев предусмотрена специальная операция – операция приведения типов. Ее синтаксис таков: type1(v), где v – переменная типа type2. Для того, чтобы можно было применять операцию приведения типов, нужно, чтобы переменные типов type1 и type2 занимали одинаковый объем памяти.

Пример 5: Применение операции приведения типов.

```
1 : uses Crt;
2 : var
3 :   s:shortint;
4 :   w:byte;
5 :   l:longint;
6 :   i1,i2:integer;
7 : begin
8 :   Clrscr;
9 :   w:=255;
10:   s:=w;{типы совместимы, но в s будет записано не 255}
11:   writeln('w = ',w);
12:   writeln('Вот что стало с s: ',s);
13: { s:=shortint(l); в таком операторе была бы выдана ошибка}
14:   i1:=1000;
```

```
15:   i2:=1000;
16:   l:=i1*i2;{вроде бы все чисто, но ...}
17:   writeln(i1,'*',i2,' = ',l);
18:   writeln('А с приведением типов все ОК');
19:   l:=longint(i1)*longint(i2);
20:   writeln(i1,'*',i2,' = ',l);
21:   readln;
22: end.
```

```
Результаты работы программы будут такие:
w = 255
Вот что стало с s: -1
1000*1000 = 16960
А с приведением типов все ОК
1000*1000 = 1000000
```

Рассмотрим все чудеса по порядку.

Переменная w типа byte. Поэтому если w=255, то при переводе в двоичную систему, получим, что w=11111111. При выполнении оператора s:=w в переменную s, как и положено, будет записано число 11111111. Но ведь s – типа shortint, а значит, первый бит в двоичной записи – знаковый. Т.к. на первом месте стоит 1, то число – отрицательное. А абсолютная величина его = 1. Поэтому в переменной s закодировано число –1.

Оператор в строке 14, если его раскомментировать, приведет к ошибке, т.к. переменные типов shortint и longint имеют разные размеры.

И, наконец, самое интересное: в строке 16 мы вроде бы все делаем правильно: 1000 входит в диапазон типа integer, 1000000 входит в диапазон типа longint. Но ошибка вот в чем: умножение двух чисел типа integer считается тоже типа integer. Поэтому после того, как при умножении 1000*1000 будет получен ответ 1000000, он сначала будет приведен к типу integer, а затем уже присвоен в переменную l. А если поступить так, как это сделано в строке 19, то ошибки не будет: числа i1 и i2 будут интерпретироваться как переменные типа longint, поэтому и результатом будет число типа longint.

7.7. Бестиповые ссылки

Обобщать подпрограммы можно и радикальнее: в TP можно передавать в подпрограммы бестиповые var-параметры (т.е. бестиповые ссылки). Для того чтобы ПК понимал при этом, какого же типа переменные, будем использовать приведение типа переменных.

Следующий пример – абсолютно аналогичен примеру 4, только вместо открытых массивов будут использоваться бестиповые ссылки.

Пример 6: Использование бестиповых ссылок для передачи массивов в процедуры.

```
1 : const
2 :   n=6;
3 :   m=12;
4 : type
```

```
5 :   GrossMas=array[1..20000] of integer; {gross большой}
6 :   Mas1=array [1..n] of integer;
7 :   Mas2=array [1..m] of word;
8 :   Mas3=array [1..6] of shortint;
9 : var
10:   A:Mas1;
11:   C:Mas3;
12:   B:Mas2;
13:   i:integer;
14: procedure SchrMas(const v;len:word);{v - бестиповая ссылка}
15: var                  {len - индекс последнего элемента массива}
16:   i:integer;
17: begin
18:    for i:=1 to len do
19:      write(GrossMas(v)[i],' ');{приведение массива v}
20:    writeln;                     {к типу GrossMas}
21: end;
22:
23: procedure EinfFulle(var v;len:word);{v - бестиповая ссылка}
24: var                  {len - индекс последнего элемента массива}
25:   i:integer;
26: begin
27:   for i:=1 to len do
28:     GrossMas(v)[i]:=i;
29: end;
30:
31: begin
32:   EinfFulle(A,n);
33:   writeln('Массив A');
34:   SchrMas(A,n);
35:   EinfFulle(B,m);
36:   writeln('Массив B до заполнения массива C');
37:   SchrMas(B,m);
38:   EinfFulle(C,6);
39:   writeln('Массив C (печать с помощью SchrMas)');
40:   SchrMas(C,6);
41:   writeln('Массив C (обычная посимвольная печать)');
42:   for i:=1 to 6 do
43:     write(C[i],' ');
44:   writeln;
45:   writeln('Массив B после заполнения массива C');
46:   SchrMas(B,m);
47:   readln;
48: end.
```

```
Результаты работы программы:
Массив A
1 2 3 4 5 6
Массив B до заполнения массива C
1 2 3 4 5 6 7 8 9 10 11 12
Массив C (печать с помощью SchrMas)
1 2 3 4 5 6
```

```
Массив C (обычная посимвольная печать)
1 0 2 0 3 0
Массив B после заполнения массива C
4 5 6 4 5 6 7 8 9 10 11 12
```

Тип GrossMas – это тип-шаблон, к которому будут приводиться все передаваемые в процедуру параметры, т.е. вне зависимости от типа переменной она будет рассматриваться как переменная типа GrossMas (точнее: как ссылка на переменную типа Grossmas).

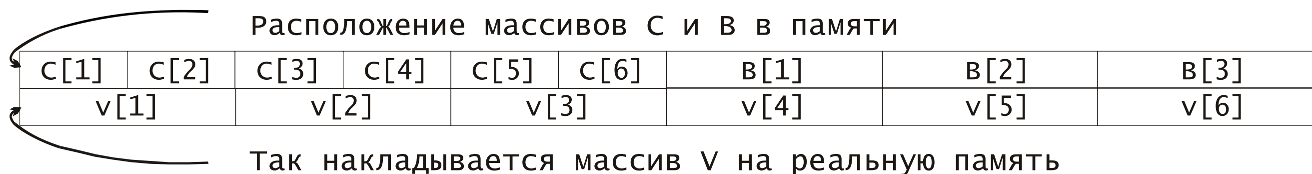
В обе процедуры, приведенные в этом примере, передается 2 параметра: сам массив (как бестиповая ссылка), и длина массива.

Теперь давайте разберемся с тем, что происходит при каждом вызове процедуры.

При вызове EinfFulle(A,n); массив A типа Mas1 будет интерпретироваться как массив типа GrossMas. Т.к. длина массива A = n, то проблем при этом не возникнет: в переменной A, трактуемой как GrossMas, будут заполнены первые n элементов числами типа integer; затем, после выхода из подпрограммы, переменная A будет рассматриваться уже как переменная типа Mas1, т.е. как массив чисел типа integer длины n. В результате таких манипуляций с типами мы ничего не напутали, и программа отработала так, как мы и хотели. При вызове процедуры SchrMas(A,n); механизм работы будет аналогичен.

Немного хитрее будет при передаче в качестве параметра массива B типа Mas2. Внутри подпрограммы будет создана ссылка A на массив B, но трактоваться она будет как ссылка на массив типа GrossMas, элементы которого типа integer (заметьте: сам массив B состоит из элементов типа word). Внутри процедуры массив заполняется элементами типа integer, которые рассматриваются после выхода из подпрограммы как элементы типа word. Поэтому, если бы массив внутри процедуры заполнялся отрицательными элементами, то при выходе из подпрограммы мы бы увидели совершенно другие результаты (см. раздел приведение типов). Но в нашей программе заполнение ведется положительными числами, и никаких проблем не возникает.

А вот с массивом C совсем беда: «квантовые эффекты» приводят к тому, что выполняется целый ряд действий, ни одно из которых не было задумано нами. Давайте разбираться! Массив C состоит из 6 переменных типа shortint, занимающих по 1 байту каждая. А внутри функции этот массив интерпретируется как массив, состоящий из элементов типа integer. Так как в функцию была передана длина массива, равная 6, то заполняться будут первые 6 его элементов, т. е. 12 байтов, а не 6, как это необходимо.



На рисунке показано, как будет расположен в памяти массив V. Понятно, что в первые два элемента массива C будет записано только лишь число 1, в 3-й и 4-й элементы – число 2, в 5-й и 6-й – число 3, а числа 4, 5, 6 будут записываться уже поверх массива B. Поэтому запортилась информация, находящаяся в массиве B, обращения к которому вообще не было. Остается еще одна проблема: как поделят элементы C[1] и C[2] одно число, которое записывается на их место. Логично было бы предположить следующее: ведущие 8 нулей в двоичной записи отойдут к числу C[1], остальные 7

нулей и единица – к элементу C[2]. В результате в C[1] должен быть записан ноль, а в C[2] единица.

c[1]								c[2]							
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Но если вы заглянете в результаты работы программы, то вы заметите прямо противоположную ситуацию: C[1]=1, а C[2]=0. А все дело вот в чем: в памяти байты хранятся в обратном порядке: если число 1 в двоичной системе имеет вид 00000000 00000001, то в памяти оно хранится так: 00000001 00000000. Именно этим и объясняется то, что числа 1, 2, 3 хранятся в обратном порядке.

Для того чтобы лучше сориентироваться в этом, слегка необычном способе хранения информации, измените в примере строку 28. Напишите вместо нее следующее:

```
GrossMas(v)[i]:=i+255;
```

Посмотрите, какие будут результаты и разберитесь, что там и как.

И еще одно: вы видите, что если бы мы не написали поэлементный вывод массива C, то могли бы даже не заметить ошибки, т.к. при печати с помощью процедуры SchrMas массив C и кусок массива B будут выводиться как целые числа.

Вы видите, что операция приведения типов при неграмотном ее использовании может привести к очень серьезным ошибкам, которые очень трудно отлаживать и исправлять.

Итак: мы достигли определенного обобщения подпрограмм, используя бестиповые ссылки, но в то же время добавился целый ворох дополнительных проблем: некоторые процедуры, например, сортировка, не зависят от того, тип данных integer или word, а вот заполнять массивы надо осторожно: диапазоны значений двух типов различны. Поэтому если вы работаете с массивами одного типа, то лучше используйте открытые массивы, - так будет и проще и надежнее. Однако не пренебрегайте бестиповыми ссылками – они нам еще сослужат добрую службу!

7.8. Процедурный и функциональный типы

В школе всем нам давали определение функции $y = f(x)$ как зависимости между значениями независимой переменной x и соответствующими значениями y. В то же время в математике часто рассматриваются зависимости более общего вида: $h(y) = F\left(y(x)\right)$, где каждой функции $y(x)$ соответствуют определенные значения $h(y)$. Такие зависимости, где независимой переменной являются не числа, а функции, обычно называют функционалами.

Примеры функционалов:

- $h(y(x)) = y(1)$, т.е. каждой функции $y(x)$ ставится в соответствие ее значение в точке $x = 1$ (предполагается, что функция $y(x)$ определена в этой точке).
- $I\left(f(x)\right) = \int\limits_a^b f(x)dx$ - интегральный функционал ($f(x)$ - интегрируема на сегменте $[a,b]$).

Замечание: если функционал $F(y)$ задан на множестве функций вида $y(x) = c$, $c \in R$, то мы фактически определяем обычную функцию, заданную на множестве R. Поэтому

обычные функции можно считать функционалами, заданными на специально подобранном множестве функций.

Раньше мы с вами писали подпрограммы, которые принимали в качестве параметров либо числа, либо массивы чисел, - т.е. с точки зрения математики мы писали функции. Но часто требуется в подпрограммы передавать другие подпрограммы, - т.е. программировать функционалы. Например, если наша цель – отсортировать массив, то необходимо написать 2 процедуры: сортировку по возрастанию, и сортировку по убыванию. При этом обе процедуры будут отличаться лишь условиями в операторе if.

Для того, чтобы передать в одну подпрограмму другую, заметим, что подпрограмма – это некоторая последовательность машинных команд. Следовательно, у подпрограммы есть адрес ее начала, а значит, можно ее передавать в качестве параметра другим подпрограммам. Для ссылок на подпрограммы используется специальный тип данных – процедурный (функциональный) тип. Как он описывается, мы рассмотрим на примере.

Пример 7: Челночная сортировка массивов любой длины по возрастанию или убыванию.

```
1 : const
2 :   n=6;
3 : type
4 :   Mas=array [1..n] of longint;
5 :   Funk=function(a,b:longint):boolean;
6 :
7 : function Mehr(a,b:longint):boolean; far;
8 : begin         {Mehr = больше}
9 :   Mehr:=a>b;
10: end;
11:
12: function Weniger(a,b:longint):boolean; far;
13: begin         {Weniger = меньше}
14:   Weniger:=a<b;
15: end;
16:
17: procedure RandFulle(var A:array of longint;k:word);
18: var
19:   i:integer;
20: begin
21:   for i:=0 to High(A) do
22:     A[i]:=random(k);
23: end;
24:
25: procedure DruckMas(var A:array of longint);
26: var          {drucken - печатать (сравните: друкувати)}
27:   i:integer;
28: begin
29:   for i:=0 to High(A) do
30:     Write(A[i],' ');
31:   writeln;
32: end;
```

```
33:
34: procedure KahnSort(var A:array of longint;F:Funk); {Kahn - чёлн}
35: var
36:   i,j:integer;
37:   tmp:longint;
38: begin
39:   for i:=0 to High(A)-1 do
40:      if F(A[i+1],A[i])=false then {нарушен ли порядок}
41:         begin
42:         tmp:=A[i];             {меняем элементы местами}
43:         A[i]:=A[i+1];
44:         A[i+1]:=tmp;
45:         for j:=i downto 1 do  {погружаем элемент}
46:           if F(A[j],A[j-1])=false then{если элемент не на месте}
47:            begin                   {то меняем его с предыдущим}
48:            tmp:=A[j-1];
49:            A[j-1]:=A[j];
50:            A[j]:=tmp;
51:            end
52:          else
53:            break;
54:         end;
55: end;
56:
57: var
58:   M:Mas;
59: begin
60:   randomize;
61:   RandFulle(M,1000);
62:   writeln('Исходный массив');
63:   DruckMas(M);
64:   KahnSort(M,Mehr);
65:   writeln('Отсортированный массив');
66:   DruckMas(M);
67: end.
```

В 5-й строке объявлен тип Funk (функция = Funktion). Типу Funk принадлежит любая функция с формальными параметрами (a,b:longint) и типом возвращаемого значения boolean.

В этом примере таких функций две: Mehr и Weniger. Заметьте, что после заголовка этих функций стоит слово far. Оно означает то, что функция будет компилироваться с расчетом на дальнюю модель памяти. Есть 2 модели памяти – ближняя и дальняя. При ближней модели памяти вызов подпрограммы выполняется быстрее, а при дальней – медленнее. Но ближние вызовы можно делать только в рамках того модуля данных, в котором описана подпрограмма, а на дальние вызовы такого ограничения нет.

Для того чтобы при вызове подпрограммы использовалась дальняя модель памяти, можно писать после названия подпрограммы директиву far (см. строки 7, 12). Если надо использовать ближнюю модель, то можно ставить директиву near (но

ближняя модель памяти используется по умолчанию, поэтому слово near писать не обязательно).

Алгоритм челночной сортировки – модификация сортировки пузырьком. Этот алгоритм – нечто среднее между сортировкой пузырьком и сортировкой вставками: массив также, как и в сортировке вставками, делится на 2 части: отсортированную и неотсортированную. На каждом шаге алгоритма выбирается новый элемент из неотсортированной части, и он начинает погружаться в отсортированный массив, на каждом шаге обмениваясь местами с предыдущим элементом, пока не доберется до своего места в отсортированной части.

Вообще говоря, этот алгоритм работает медленнее, чем сортировка вставками, т.к. процесс погружения в челночной сортировке занимает больше времени. Я выбрал эту сортировку только для того, чтобы вы не решили, что я халтурщик: уже и так написал 2 примера, делающие одно и то же, так еще и сортировки будет писать по второму кругу.

Для нас главное следующее: в процедуре KahnSort, кроме самого массива есть еще один параметр: функция типа Funk, цель которой задавать порядок расположения элементов: в строках 40 и 46 программа проверяет, нарушен ли порядок следования элементов. Если в качестве функции F будет передана функция Mehr, то функция F будет проверять, не является ли первый аргумент больше второго, если же Weniger, - то, наоборот, будет ли первый меньше второго.

Справедливости ради следует заметить, что сортировать можно не только по убыванию или возрастанию, а, например, по убыванию суммы цифр, - тогда нам достаточно написать функцию, проверяющую, будет ли сумма цифр первого числа меньше суммы второго, и передать ее в качестве параметра в функцию сортировки.

Заметьте, мы начали главу с простой целью: научиться разбивать большую программу на небольшие кусочки, а при этом получили мощный механизм, позволяющий писать гораздо более абстрактные и универсальные программы. Но это лишь начало: в следующей главе вы столкнетесь с гораздо более удивительными применениями подпрограмм, которые могут существенно изменить ваши представления о программировании.

Задачи

1. Написать функции tg(x), ctg(x), a^x, chx, shx, $\log_a b$, arcsin(x), arccos(x), arcctg(x).
2. Написать функции {x}, [x] (только правильно работающие!)
3. Написать функцию вычисления скалярного произведения массивов.
4. Написать функцию нахождения индекса минимального элемента в массиве.
5. Написать функцию нахождения индекса заданного числа в массиве. Если таких чисел в массиве несколько, то возвращайте индекс любого из них. Если заданного числа в массиве нет, то возвращать 0.
6. Раньше у нас был пример программы, вычисляющей решения линейного уравнения. Найдите его, и напишите функцию, которая бы выполняла аналогичные действия.
7. Написать процедуру, которая бы выводила все делители заданного числа.
8. Даны все стороны треугольника. Найти его площадь и периметр. (Вычисление площади и периметра должны быть отдельными функциями).

9. Функция, которая во заданному числу возвращает его “перевертыш”. (Например: 1235 – 5321).
10. Последовательность Пупкина образуется следующим образом: первое число последовательности – единица, далее к уже записанным числам дописываются они же, но все нули заменяются на единицы и наоборот (начало последовательности: 10010110…). Напишите функцию, выдающую n-ю цифру последовательности Пупкина.
11. В банкомате есть купюры только в 3 и 5 тугрика. Ваша цель – написать процедуру, которая по введенной сумме (целому числу) печатает 2 числа - количество купюр достоинством в 3 и 5 тугриков соответственно, или –1, если введенную сумму выдать такими купюрами нельзя.
12. Теперь решите предыдущую задачу, если одна купюра достоинством в 3 тугрика а вторая -любое число k, которое больше 3, и не делится на 3.
13. Решите предыдущую задачу, если есть купюры по x и y тугриков.
14. У вас есть монеты достоинством в 1, 2, 5, 10, 25, 50 копеек. Напишите программу, которая любую введенную сумму выдавала бы минимальным количеством монет. Алгоритм должен быть оптимален.
15. Обобщите предыдущую задачу: пусть у вас в распоряжении есть монеты достоинством в $a_1, a_2, ..., a_n$. Можно ли просто перенести результаты задачи 14 на общий случай?
16. Челноку на 7-м километре заказали партию в N пар носков. В Турции, где он закупает товар, ему предложили скидки в 5% за каждую полную связку (12 пар носков) и за каждую коробку (12 связок) 15%. Челноку пришло в голову, что хотя лицензии на розничную торговлю у него нет и не предвидится, а заказчик бОльшую партию носков купить откажется, но ему может оказаться выгоднее купить больше носков и часть выкинуть, чем покупать в точности заказанную партию носков. Помогите челноку подсчитать, сколько носков ему надо купить, чтобы и заказ выполнить и заплатить как можно меньшую сумму: напишите функцию, которая по введенному числу N возвращает кол-во носков, которые челнок должен купить.
17. Пусть $n \in N$. Функция Эйлера $\varphi(n)$ равна количеству значений из множества $\{0, 1, 2, ..., n-1\}$, взаимно простых с n. Например, $\varphi(1) = 1$, $\varphi(2) = 1$, $\varphi(3) = 2$, … Покажите, что если p - простое число, то $\varphi(p) = p - 1$, и вычислите $\varphi\left(p^e\right)$, где $e > 0$.
18. Функция $f(n)$ называется мультипликативной, если $f(rs) = f(r) f(s)$. Докажите, что произведение двух мультипликативных функций является мультипликативной функцией.
19. Докажите, что функция Эйлера (см. упражнение 17) мультипликативна. На основе этого предложите простой способ вычисления $\varphi(n)$, если n разложено на простые множители.
20. **(!)** Напишите функцию, которая бы вычисляла приближенное значение $\int_a^b f(x)dx$.

    Функция должна принимать в качестве одного из параметров ссылку на функцию $f(x)$.

Проект 1: Длинная Арифметика

Очень часто, например, в криптографии, надо использовать большие целые числа. В TP наибольший тип – longint – слишком мал, а вещественные типы данных принципиально не подходят, т.к. все вычисления должны производиться без потери точности. Поэтому надо создать собственный тип данных, который позволит обрабатывать гигантские натуральные числа без потери точности.

Представлять числа будем в виде массива, причем:

```
const
  MaxZiff=300;{Максимальное количество разрядов (Radix-ичных)}
  Radix=1000;{основание системы счисления}
type
  LangeZ = array [0..MaxZiff] Of integer;   {lange Zahl - длинное
число}
```

В A[0] будет находиться количество Radix-цифр в числе, а дальше «цифры» числа в обратном порядке, например:

$12345678910=12\cdot1000^3+345\cdot1000^2+678\cdot1000^1+910\cdot1000^0$

Значит, представление чисел будет такое:

4	910	678	345	12

Это представление удобно для выполнения арифметических действий с числами, поэтому не думайте, что оно сильно мудреное.

Вашей целью будет написать такие подпрограммы:

1. сложение чисел
2. вычитание чисел (если число, от которого отнимают меньше, чем число, которое отнимают, то вычисления не проводите, а сразу пишите, что операция невыполнима, т.к. числа – натуральные и результат не должен быть отрицательным).
3. умножение чисел
4. вычисление остатка от деления (аналог mod)
5. вычисление неполного частного (аналог div)
6. вычисление НОД 2-х чисел.

Т.к. числа большие и в стандартные типы данных не влезают, а пользователю неудобно вводить число в виде массива, то мы будем считывать с клавиатуры число в виде строки (тип string), и затем переводить его из строки в LongI.

Тип string – это строковый тип. Мы его не проходили, поэтому я написал для вас подпрограммы перевода числа в строку и обратно, а также несколько других вспомогательных подпрограмм. Вводить строки можно с помощью read и readln, а печатать с помощью write и writeln.

Процедура AddLangeZ складывает 2 длинных натуральных числа, поэтому пункт 1 уже выполнен.

Пример 1: Вспомогательные функции для «длинной арифметики».

```
1 : uses
2 :   Crt;
3 : const
```

```
4 :   MaxZiff=300;{Максимальное количество разрядов (Radix-ичных)}
5 :   Radix=1000;{основание системы счисления}
6 : type
7 :   LangeZ = array [0..MaxZiff] Of integer;   {Длинное число}
8 :
9 : procedure Nullieren(var A:LangeZ); {Обнуление числа}
10: var
11:   i:integer;
12: begin
13:   for i:=1 to MaxZiff do
14:     A[i]:=0;
15:   A[0]:=1;
16: end;
17:
18: {Записывает в A число, хранящееся в строке Int}
19: procedure StellLangeZ(var Int:string;var A:LangeZ);
20: var
21:   i,j:byte;
22: begin
23:   Nullieren(A);
24:   for i:=1 to length(Int) do
25:     begin
26:     for j:=A[0] downto 1 do
27:       begin
28:       A[j+1]:=A[j+1]+ ((A[j]*10) div Radix);
29:       A[j]:=(A[j]*10) mod Radix;
30:       end;
31:     A[1]:=A[1]+ord(Int[i])-ord('0');
32:     if (A[A[0]+1]>0) then
33:       inc(A[0]);
34:     end;
35: end;
36:
37: {Переводит длинное число A в строковое представление}
38: function LangeZZurZeile(Var A:LangeZ):string;
39: var
40:   s,s1,tmp:string;
41:   i,k:byte;
42: Begin
43: {В tmp кол-во символов = кол-ву десятичных цифр в Radix div 10}
44:   Str(Radix div 10,tmp);
45:   Str(A[A[0]],s);
46:   for i:=A[0]-1 downto 1 do
47:     begin
48:     str(A[i],s1);
49:     for k:=1 to (length(tmp)-length(s1)) do
50:       s:=s+'0';
51:     s:=s+s1;
52:     end;
53:   LangeZZurZeile:=s;
54: end;
55:
```

```
56: {KopiereLangeZ(A,B) это просто B:=A}
57: procedure KopiereLangeZ(const A:LangeZ;var B:LangeZ);
58: var
59:   i:integer;
60: begin
61:   Nullieren(B);
62:   for i:=0 to A[0] do
63:     B[i]:=A[i];
64: end;
65:
66: {Ввод числа A с клавиатуры (заканчивается нажатием Enter}
67: procedure LeseLangeZ(var A:LangeZ);
68: var
69:   s:string;
70: begin
71:   Nullieren(A);
72:   readln(s);
73:   StellLangeZ(s,A);
74: end;
75:
76: {C:=A+B}
77: procedure AddLangeZ(const A,B:LangeZ;var C:LangeZ);
78: var i:byte;
79:     k:longint;
80: begin
81:   Nullieren(C);
82:   if A[0]>B[0] then
83:     k:=A[0]
84:   else
85:     k:=B[0];
86:   for i:=1 to k do {Складываем столбиком разряды}
87:     begin       {Помните: разряды хранятся в обратном порядке}
88:     C[i]:=A[i]+B[i]+C[i];
89:     if C[i]>=Radix then {Если перешли на старший разряд}
90:       begin
91:       inc(C[i+1]);
92:       C[i]:=C[i]-Radix;
93:       end;
94:     end;
95:   if C[k+1]>0 then {Устанавливаем номер старшего разряда}
96:     C[0]:=k+1
97:   else
98:     C[0]:=k;
99: end;
100:
101: var
102:   A,B,C:LangeZ;
103:   s1,s2:string;
104: BEGIN
105:   ClrScr;
106:   s1:='123334';
107:   s2:='9432441';
```

```
108:
109:   StellLangeZ(s1,A);
110:   writeln(LangeZZurZeile(A));
111:   StellLangeZ(s2,B);
112:   writeln(LangeZZurZeile(B));
113:   AddLangeZ(A,B,C);
114:   writeln(LangeZZurZeile(C));
115:   readln;
116: End.
```

Глава 8: Рекурсия

Вы уже знаете, что программу можно разбивать на подпрограммы и вызывать их из основной программы, или друг из друга. На протяжении всей предыдущей главы мы обходили особый случай: что произойдет, если подпрограмма будет вызывать в процессе работы саму же себя? Такие подпрограммы называются рекурсивными. Рекурсия очень важна для программирования, поэтому мы рассмотрим эту тему достаточно подробно. Но прежде я хотел бы немного рассказать о том, где еще можно встретить рекурсию.

8.1. Рекурсия в биологии

Рассмотрим процесс митотического деления клетки. Деление клетки начинается с ядра: образуются хромосомы, занятая ядром область становится более вытянутой, хромосомы делятся на 2 половинки, и они разбегаются в противоположные части этой области, где собираются в том же количестве, что и исходные хромосомы до начала деления. Затем на каждом из полюсов они окружаются мембраной, и теряют четкие контуры. Так образуются 2 новых ядра, идентичных исходному. Между ними появляется перегородка, разделяющая цитоплазму и другие компоненты клетки на равные части. В результате этого процесса, называемого митозом (греч. *mitos* - «нить»), образуются 2 клетки, содержащие ту же генетическую информацию, что и материнская клетка.

Если рассматривать митоз на протяжении нескольких поколений, то налицо его рекурсивный характер: родительская клетка порождает 2 дочерние, затем каждая из них также «вызывает процедуру размножения», делясь еще на 2 клетки и. т. д.

Рекурсивный характер размножения характерен не только для живых организмов – существуют органические макромолекулы, способные синтезировать свои точные копии, - их называют репликативными молекулами, или репликаторами.

А теперь вспомните, как размножаются гидры – новые особи «отпочковываются» от тела матери. В результате получается оригинальный организм: на одной гидре находится много гидр меньших размеров, а на них уже начинают появляться крохотные гидрочки. Чем не рекурсия?!

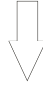
8.2. Рекурсия в литературе

Структура предложения

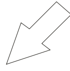
Сложные предложения имеют чисто рекурсивный характер: в главном предложении можно выделить ряд вложенных предложений, выделяющихся в русском языке запятыми, что выгодно отличает его от английского, которые могут также состоять из нескольких предложений, каждое такое «подподпредложение» - еще из нескольких и т. д. Структура предыдущего предложения приведена на рис. 8.1.

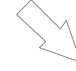
В природе аналогичной структурой обладают деревья, поэтому в программировании рекурсивные структуры данных, подобные описанным выше, носят название деревьев.

Сложные предложения имеют чисто рекурсивный характер: X



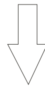
X = в главном предложении можно выделить ряд вложенных предложений, Y, которые могут в свою очередь также состоять из нескольких предложений, Z





Y = выделяющихся в русском языке запятыми, F

Z = каждое такое «подподпредложение» - еще из нескольких и т. д.



F = что выгодно отличает его от английского

Рис 8.1. Структура предложения

Рекурсивный сон

Некоторые люди могут видеть сны во сне; считается, что это является следствием плохого психического самочувствия. Действительно, как-то не по себе становится, когда просыпаешься и думаешь, что вернулся в реальность, но на самом деле это все еще сон, только меньшего уровня вложенности. Очень редко бывают сны третьего порядка. В литературе часто встречаются описания рекурсивных снов. Я приведу отрывок из рассказа Станислава Лема – «Сказка о короле Мурдасе»:

«Впал он из сна в сон – новый, снящийся предыдущему, а тот еще более раннему пригрезился, так что этот, теперешний, был уже будто третьей степени; уже совершенно явно все оборачивалось тут изменой, пахло отступничеством; знамена, словно перчатки, из королевских на изнанку черную выворачивались, ордена были с резьбой, словно шеи обезглавленные, а из сверкающих золотом труб не музыка боевая звучала, но дядин смех громыхал ему на погибель. Взревел король гласом иерихонским, кликнул войско – пусть хоть пиками колют, только бы разбудили! Ущипните! – требовал он громогласно. И снова: Яви мне!!! Яви!!! – впустую; и опять из цареубийственного, крамольного сна пытался он пробиться в коронный, но расплодилось в нем снов, что собак, шныряли они повсюду, словно крысы, ширился всюду кошмар, как чума, разносилось по городу- тишком, полушепотом, втихомолку, украдкой – неведомо что, но такое ужасное, что не приведи господь!»

8.3. Рекурсия в математике

Если последовательность задана так, что с некоторого шага каждое ее следующее значение выражается через предыдущие, то такое задание последовательности называется рекуррентным (то же самое, что и рекурсивным, но математикам так нравится больше).

Классический пример – последовательность Фибоначчи : $f_1 = f_2 = 1$, $f_n = f_{n-1} + f_{n-2}$, $n > 2$.

Другой пример – полиномы Чебышева: $P_0 = 1$, $P_1 = x$; $P_{n+1}(x) = 2xP_n(x) - P_{n-1}(x)$, $n > 2$.

Рекуррентные соотношения зачастую (хотя далеко не всегда) гораздо удобнее итерационных.

Например, явная формула для чисел Фибоначчи такова (формула Бине):

$$f_n = \frac{1}{2^n\sqrt{5}}\left(\left(1+\sqrt{5}\right)^n - \left(1-\sqrt{5}\right)^n\right)$$

Формула для полиномов Чебышева:

$$P_n(x) = \frac{1}{2^n}\left(\left(x-\sqrt{x^2-1}\right)^n + \left(x+\sqrt{x^2-1}\right)^n\right)$$

Ясно, что в таких случаях использование рекуррентных соотношений вполне оправдано, а выражения в явном виде могут понадобиться разве что в теории.

8.4. Зачем нужна рекурсия в программировании?

Единственный подход, который мы использовали до этой главы, был итерационный, в котором основными понятиями были цикл и условный оператор. For'ы, while'ы и repeat'ы верой и правдой служили нам при решении как простых, так и более сложных задач (таких как сортировки и поиск данных). Но существуют задачи, реализация которых с помощью итерационных алгоритмов – нелегкий труд, а с помощью рекурсии можно написать простое и изящное решение. Задача о ханойских башнях – классический тому пример.

Ханойские башни

Представьте себе древневосточный храм, в котором находится бронзовая пластина с тремя алмазными стержнями, каждый – длиной в один локоть и толщиной с тело пчелы. На один из стержней Брахма при сотворении мира нанизал 64 диска из чистого золота так, что они образуют пирамиду. День за днем, год за годом жрецы-брахманы переносят диски с одного стержня на другой, следуя законам Брахмы: перекладывать можно только по одному диску и больший по размерам диск нельзя класть на меньший. Когда брахманы перенесут таким образом все диски, то жрецы, башня и храмы превратятся в пепел и наступит конец света.

Такова древняя легенда.

Нашей целью будет написать программу, которая имитировала бы работу брахманов, т.е. печатала бы последовательность действий, с помощью которой можно переставить все диски с одного колышка на другой. Временно оставим мысли о страшных последствиях нашего занятия для человечества (как вы убедитесь, все не так плохо), а сосредоточимся на написании алгоритма.

Для определенности мы будем переносить диски со стержня А на стержень С, а число дисков = n.

Самый изящный алгоритм такой:

1. Если $n>1$, то
   - Переместить $n-1$ диск с колышка А на колышек В, используя стержень С как вспомогательный
   - Переместить диск с колышка А на колышек С
   - Переместить $n-1$ диск с колышка В на колышек С, используя стержень А как вспомогательный
2. Если $n=1$, то переместить диск с колышка А на стержень С.

Вы видите, что хотя мы не знаем общего алгоритма решения задачи (имеется в виду алгоритм решения с помощью цикла), но мы можем выразить решение задачи для n дисков в терминах $n-1$ диска, которая, в свою очередь, сводится к задаче для $n-2$

дисков, и т. д., пока задача не сведется к перемещению 1 диска, которую мы легко можем решить.

Реализацию этого алгоритма на ТР рассмотрим несколько позднее, а вы можете попытаться придумать итерационный алгоритм решения задачи о Ханойских башнях (упражнение № 6).

Рисование фракталов

На рис. 8.2. вы видите снежинку Коха после 0, 1, 2, 5 итераций. Исходная фигура – треугольник. На первом шаге на каждой стороне треугольника удаляется третья часть, и достраивается вместо нее две стороны, как показано на рисунке. На втором шаге с каждой стороной многоугольника проделывается то же самое и т.д. После бесконечного числа итераций получим кривую, которая называется снежинкой Коха. Как нарисовать такое изображение (приближенно, естественно), зная лишь длину стороны исходного треугольника?

С одной стороны, описывать структуру снежинки с помощью формулы – безнадежное дело, но в тоже время она поражает своей простотой и изяществом: любая часть снежинки подобна целому! В будущем мы с вами познакомимся с методом построения снежинки Коха и многих других фрактальных узоров.

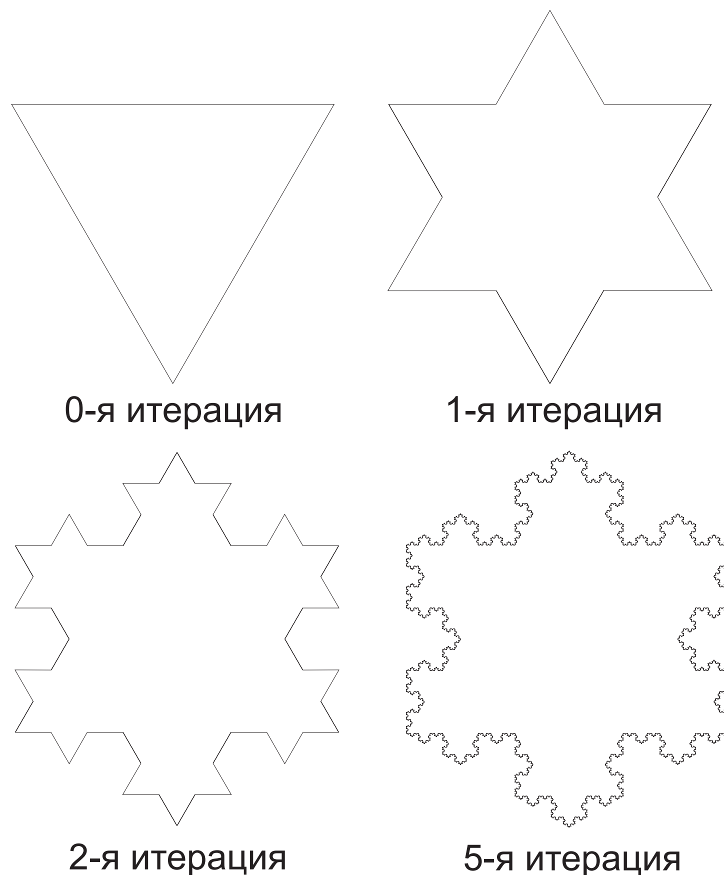


Рис 8.2. Снежинка Коха

Теперь, когда вы начали осознавать великий смысл рекурсии, рассмотрим несколько примеров.

8.5. Алгоритм Евклида

В части, посвященной циклам, мы рассмотрели алгоритм, позволяющий находить НОД двух целых чисел. Теперь мы напишем алгоритм Евклида с помощью рекурсии.

Из обоснования алгоритма Евклида следует, что

$$НОД(a,b) = \begin{cases} НОД(b,q), q \neq 0 \\ b, q = 0 \end{cases}$$, где $q = a \bmod b$.

Пример 1: Алгоритм Евклида.

```
1 : var
2 :   a,b:longint;
3 :
4 : function GGT (a,b:longint):longint;
5 : var
6 :   q:longint;
7 : begin
8 :   q:=a mod b;
9 :   if q<>0 then
10:     GGT:=GGT(b,q) {Рекурсивный вызов}
11:   else      {Если q=0, то b=НОД(a,b)}
12:     GGT:=b;
13: end;
14:
15: BEGIN
16:   writeln('Введите 2 натуральных числа');
17:   readln(a,b);
18:   writeln('НОД этих чисел (рекурсия): ',GGT(a,b));
19:   readln;
20: END.
```

Продемонстрируем ход вычислений для нахождения НОД(54,15).

Уровень рекурсии	Рекурсивный спуск	Рекурсивный возврат
1	a=54;b=15;q=9; GGT:=GGT(15,9)	GGT:=3;
2	a=15;b=9;q=6; GGT:=GGT(9,6)	GGT:=3;
3	a=9;b=6;q=3; GGT:=GGT(6,3)	GGT:=3;
4	a=6;b=3;q=0; GGT:=3	

После вызова функции GGT(54,15) будет вычислено значение q=9. Так как оно не равно 0, то НОД(54,15)=НОД(15,9) и функция GGT будет вызвана еще 1 раз (при этом переменной GGT еще не присвоено ее значение). На втором уровне рекурсии будет создана новая переменная GGT (локальная для данной функции и никоим образом не связанная с переменной GGT из предыдущего вызова подпрограммы), и в нее, так как $q \neq 0$, будет присвоено значение функции GGT(9,6). На 3-ем шаге рекурсии действия будут аналогичны. А когда будет вызвана функция GGT(6,3), то остаток будет равен 0, и дальнейших вызовов рекурсии не последует, а в переменную GGT (4-ю по счету) будет записано значение 3. Теперь начинается обратный процесс, называемый рекурсивным возвратом: значение последнего вызова функции GGT будет возвращено

в переменную GGT, которая была создана на 3-ем шаге рекурсии, затем функция GGT 3-го уровня вложенности закончит свою работу и вернет результат, и т.д. В конце результат будет возвращен в основную программу, и распечатан на экран.

Замечание: Не забывайте ставить выход из рекурсии (в данном примере – ветвь else).

Если рекурсивный способ нахождения НОД вам показался сложнее, чем итерационный, то это лишь потому, что вы уже привыкли к циклам. Даже беглого взгляда на процедуру достаточно, чтобы заметить элегантность решения. Лишь один условный оператор, и задача решена.

8.6. Вычисление натуральной степени числа

Следующий пример – вычисление натуральной степени числа. Реализуем наиболее простой алгоритм: будем пользоваться обычным определением:

$$\begin{cases} a^n = a \cdot a^{n-1}, n > 0 \\ a^0 = 1 \end{cases}$$

В качестве вещественного типа в этом примере я выбрал тип extended. Если вы уже забыли, что в TP кроме типа real есть также другие вещественные типы (я бы наверняка забыл), то напомню, что extended – это тип данных с наибольшим диапазоном значений и занимает он в памяти 10 байт. Для того, чтобы его можно было использовать, надо подключить математический сопроцессор. Это делается директивой {$N+} (подробнее о директивах компилятора можете посмотреть в Help'е). Все остальное – аналогично тому, что было в предыдущем примере.

Пример 2: Вычисление степени (Potenz) числа с помощью рекурсии.

```
1 : {$N+} {подключаем арифметический сопроцессор}
2 : function Potenz(a:extended;k:integer):extended;
3 : begin
4 :   if (k<=0) then
5 :     Potenz:=1    {a^0=1}
6 :   else
7 :     Potenz:=a*Potenz(a,k-1); {a^n=a*a^(n-1)}
8 : END;
9 :
10: BEGIN
11:   writeln('Рекурсивная степень',Potenz(2,100):10:8);
12:   readln;
13: END.
```

8.7. Приближенное решение уравнений методом бинарного деления

В школе на уроках математики всех нас учили виртуозно решать тригонометрические, показательные, логарифмические уравнения. Они, как правило, допускали точное решение, которое и надо было найти. Но школьники старших классов уже знают, что даже обычные алгебраические уравнения 5-й степени и выше в общем случае не могут быть решены точно. Поэтому надо применять приближенные методы решения уравнений. Простейший из алгоритмов - метод бинарного деления (бисекции).

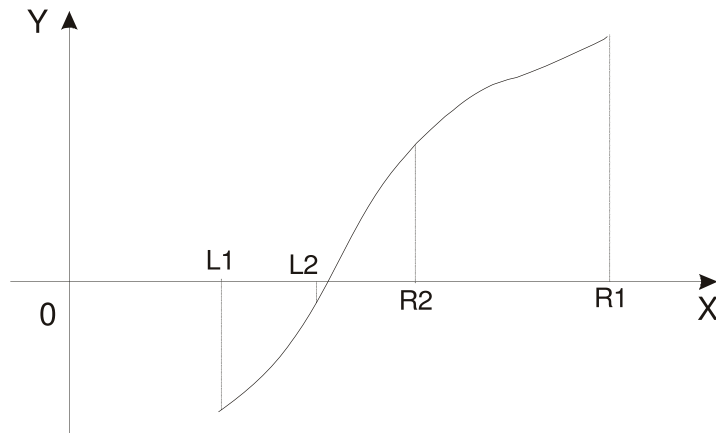


Рис 8.3. Решение уравнений методом бисекции

Пусть дано уравнение вида $f(x)=0$, имеющее на промежутке $[a,b]$ ровно 1 корень. $f(x)$ предполагается непрерывной и монотонной на сегменте $[a,b]$ функцией. Пусть для определенности $f(x)$ будет возрастающей.

Так как точное решение получено быть не может, то надо задать точность вычислений ε. Это означает, что если $|f(x)|<\varepsilon$, то x можно считать корнем уравнения $f(x)=0$.

Тогда метод нахождения корня будет таков:

Пусть l, r - соответственно левая и правая границы сегмента. Тогда вычисляем значение $f\left(\frac{l+r}{2}\right)$. Если $\left|f\left(\frac{l+r}{2}\right)\right|<\varepsilon$, то $\frac{l+r}{2}$ - приближенное значение корня уравнения. В противном случае если $f\left(\frac{l+r}{2}\right)<0$, то корень лежит в сегменте $[\frac{l+r}{2},r]$, если $f\left(\frac{l+r}{2}\right)>0$, то в сегменте $[l,\frac{l+r}{2}]$. То есть мы за один шаг уменьшили промежуток, в котором содержится корень в два раза. На следующем шаге предпринимаем аналогичные действия к новому сегменту. Ясно, что в результате таких действий мы на некотором шаге получим приближенное значение корня.

Пример 3: Решение уравнения методом бисекции (для функции, которая является непрерывной и монотонно возрастающей на сегменте, содержащем корень).

```
1 : type                        {Функциями такого вида являются}
2 :   Funk=function(x:real):real;   {левые части уравнений}
3 : var
4 :   x,y,left,right,eps:real;
5 :
6 : function Funk1(x:real):real;far;
7 : begin
8 :   Funk1:=exp(x)-x*x;
9 : end;
10:
11: function GleichLosung(left,right,eps:real;f:Funk):real;
12: var
13:   k,s:real;
14: begin
```

```
15:   k:=(left+right)/2;
16:   s:=f(k);
17:   if (abs(s)<eps) then
18:     begin
19:     GleichLosung:=k;
20:     exit;
21:     end;
22:   if (f(k)>0) then
23:     GleichLosung:=GleichLosung(left,k,eps,f)
24:   else
25:     GleichLosung:=GleichLosung(k,right,eps,f);
26: end;
27:
28: begin
29:   writeln('Введите сегмент,в котором находится корень');
30:   readln(left,right); {Вводим левую и правую границы}
31:   writeln('Введите погрешность вычислений');
32:   readln(eps);
33:   x:=GleichLosung(left,right,eps,Funk1);
34:   writeln('Приближенный корень уравнения: x = ',x:10:8);
35:   writeln('Значение функции exp(x)-x*x при этом x:
',Funk1(x):10:8);
36:   readln;
37: end.
```

Приведем теперь решение задачи о Ханойских башнях, алгоритм решения которой был рассмотрен выше.

Пример 4: Ханойские башни.

```
1 : {A - колышек, откуда переносим, C - тот, куда переносим}
2 : procedure Hanoi(n,A,C,B:byte);
3 : begin
4 :   if n=1 then
5 :     writeln(A,' -> ',C)
6 :   else
7 :     begin
8 :     {переносим со стержня A на стержень B n-1 диск}
9 :     Hanoi(n-1,A,B,C);
10:     {переносим с A на C наибольший диск}
11:     writeln(A,' -> ',C);
12:     {переносим со стержня B на стержень C n-1 диск}
13:     Hanoi(n-1,B,C,A);
14:     end;
15: end;
16:
17: var
18:   q:byte;
19: begin
20:   write('Введите количество дисков: ');
21:   readln(q);
22:   Hanoi(q,1,3,2);
```

```
23: end.
```

Результатом программы будет последовательность действий, которые надо сделать для переноса всех дисков. Причем процедура без begin-endов, заголовка и комментариев занимает всего 6 строк!!!

8.8. Рекурсия contra Итерация

Достоинство рекурсивных алгоритмов в их простоте и естественности, однако работают они медленнее итерационных, так как тратится время на рекурсивные вызовы подпрограмм (надо сохранять адрес области памяти, откуда была вызвана подпрограмма а также выделять и освобождать память под локальные переменные). Другая проблема в рекурсии – это необходимость хранения большого количества переменных в памяти (вспомните количество переменных a, b, GGT, которые были созданы в ходе работы рекурсивного Алгоритма Евклида).

Поэтому общие рекомендации такие: если алгоритм проще и естественнее написать с помощью итерационных алгоритмов (как, например вычисление факториала), то надо использовать циклы. Если алгоритм – рекурсивен по своей природе (как Ханойские башни), - то не ломайте мозги над поиском итерационного алгоритма. Даже если вы его придумаете, то его реализация будет сложнее, и вы потратите уйму времени на отладку программы, а затем, если понадобится вносить изменения, - то еще и на разбор всего того, что вы уже написали. Кроме того, гармония должна быть во всем, и в программах в том числе.

Наконец, третий случай: если вы можете без проблем написать и рекурсивное и итерационное решения задачи (как в случае алгоритма Евклида), то если подпрограмма должна работать быстро, или она очень часто используется, то лучше написать решение с помощью циклов, в противном случае вы вполне можете использовать рекурсию.

Это были практические рекомендации. Если подойти к проблеме «рекурсия contra итерация» с теоретической точки зрения, то налицо важнейший факт: т.к. циклы выполняют одни и те же действия некоторое число раз, меняя лишь некоторые параметры (напр., счетчик), то любой цикл можно заменить рекурсивной подпрограммой, параметрами которой могут быть величины, меняющиеся в цикле (хотя можно взять и иной набор), а значит – любой итерационный алгоритм можно написать с помощью рекурсии и **операторы циклов – не обязательны для языков программирования**. Можно было бы сразу после изучения оператора if переходить к функциям и рекурсии, а о циклах даже не вспоминать. Более того, есть языки программирования, в которых нет не только циклов, но даже if и оператора присваивания, например Prolog, о котором я упоминал в главе 0.

Правда, не всегда рекурсивное решение красивее. Сортировка пузырьком – яркий пример тому.

Пример 5: Сортировка пузырьком без циклов.

```
1 : uses Crt;
2 : const
3 :   n=5;
4 : type
5 :   Mas=array[1..n] of integer;
```

```
6 : var
7 :   A:Mas;
8 :
9 : procedure RandMass(var A:Mas;k:integer);
10: var
11:   i:integer;
12: begin
13:   for i:=1 to n do
14:     A[i]:=random(k);
15: end;
16:
17: procedure SchreibMass(var A:Mas);
18: var
19:   i:integer;
20: begin
21:   for i:=1 to n do
22:     write(A[i],' ');
23: end;
24:
25: procedure Auftauchen(var A:Mas;indEl:integer;LetzEl:integer);
26: var                           {auftauchen = всплывать}
27:   tmp:integer;
28: begin
29:   if indel<LetzEl then
30:     begin
31:     if (A[indel]>A[indel+1]) then
32:       begin
33:       tmp:=A[indel];
34:       A[indel]:=A[indel+1];
35:       A[indel+1]:=tmp;
36:       end;
37:     Auftauchen(A,indel+1,letzEl);
38:     end
39:   else  {Всплытие закончилось}
40:     exit;
41: end;
42:
43: {Внешний цикл в процедуре сортировки}
44: procedure BlaseSort(var A:Mas;LetzEl:integer);
45: begin
46:   if LetzEl<>1 then
47:     begin
48:     {Проводим всплытие (начиная с 1-го элемента)}
49:     Auftauchen(A,1,LetzEl);
50:     {Сортируем подмассив [1..LetzEl-1]}
51:     BlaseSort(A,LetzEl-1);
52:     end
53:   else {Массив отсортирован}
54:     exit;
55: end;
56:
57: begin
```

```
58:   Clrscr;
59:   randomize;
60:   RandMass(A,55);
61:   SchreibMass(A);
62:   writeln;
63:   BlaseSort(A,n);
64:   SchreibMass(A);
65:   readkey;
66: end.
```

Видно, что каждый цикл итерационного алгоритма записывается отдельной рекурсивной процедурой. Сортировка выполняется в процедуре BlaseSort (Blase - пузырь), в которой 2 параметра – сам массив и номер элемента, до которого будут сортироваться элементы массива. Обращу ваше внимание на то, что сам 2-й параметр нужен непосредственно для реализации самой процедуры, а увеличение функциональности самой процедуры сортировки – лишь следствие. Процедура Auftauchen – аналог вложенного цикла в итерационном алгоритме: она проводит 1 этап всплытия элементов, начиная с индекса indEl и до конца неотсортированной области, который задается индексом LetzEl.

Т.к. производительность компьютеров постоянно растет, то преимущество итерационных алгоритмов для задач, не связанных с большой вычислительной работой, вообще меркнет. Кроме того, раньше существовали языки, не поддерживающие рекурсию точно так же, как Prolog не поддерживает итерацию. Но они были постепенно вытеснены, так как их возможности были существенно ниже мощности остальных языков программирования, а логические и функциональные языки здравствуют и по сей день и некоторые задачи позволяют решать проще, чем процедурные языки и их объектно-ориентированные расширения.

Не означает ли это конец итерационных алгоритмов? Этот вопрос перекликается с другим: насколько универсальным должен быть язык программирования?

Основных ответов на второй вопрос два:

1. Языки должны быть специализированы: они должны быть оснащены компактным набором средств (часто говорят о небольшом количестве ключевых слов), которые должны быть нацелены на как можно более эффективное решение некоторых задач.
2. Языки программирования должны быть универсальны: они должны позволять программисту решать как можно более широкий круг задач, а набор средств совсем не обязательно должен быть спартанским.

Разумеется, могут быть и промежуточные точки зрения, но для рассмотрения достаточно и этих двух. Представители 1-й группы называют языки, удовлетворяющие 2-му требованию «болотом» (я лично это слышал), и приводят действительно очень изящные программы, которые имеют полное право назваться шедеврами программистского мастерства.

Прежде, чем высказать мое мнение, отмечу, что в живой природе есть виды, которые очень хорошо приспособились к определенным условиям среды, а есть те, которые выживают в разных условиях. При определенных условиях среды специалисты могут процветать быстрее, чем универсалы, однако с точки зрения эволюции универсалы устойчивее, и вымирают такие виды реже, чем специалисты. В истории тоже известны общества, которые придерживались какой-то конкретной концепции развития и небольшого набора правил, например, Спарта и Османская

империя. Неспособность этих цивилизаций объективно реагировать на изменения во внешнем мире привела к замиранию общественно-политической жизни и, в конце концов, к краху.

Лично я считаю, что и высокоспециализированные языки программирования ждет та же участь. Но в процессе их развития будут выработаны многие приемы, которые облегчат создание и увеличат возможности универсальных языков. Совершенно очевидно, что постепенно языки программирования еще больше приблизятся к естественному языку, а значит и циклы и рекурсия должны оставаться, так как многие задачи человек решает, комбинируя оба метода. Например, когда я выхожу на стадион, чтобы побегать, то часто ставлю задачу рекурсивно: пробегу-ка я половину намеченной дистанции, а там уже и до конца недалеко. А во время бега считаю круги, т.е. действую итеративно.

8.9. Быстрая сортировка

Худшее время работы алгоритма быстрой сортировки cn^2, как и у сортировки пузырем, выбором и вставками. Это означает, что можно придумать такие массивы, которые быстрая сортировка может упорядочить лишь за cn^2 операций. Однако в среднем этот алгоритм работает очень быстро – даже быстрее, чем алгоритм сортировки слиянием, который всегда работает за время $cn\log_2 n$.

Алгоритм быстрой сортировки заключается в следующем:

Выбираем любое число из массива (обычно берут средний элемент, хотя можно выбирать элемент случайным образом), и переформировываем массив на две части так, чтобы в одной части стояли все элементы, меньшие этого числа, а во второй – большие. Затем к каждой части применяем алгоритм быстрой сортировки.

Разбивать массив на 2 части, в одной из которых находятся элементы большие заданного числа x, а в другой – меньшие его, мы умеем (см. главу «Массивы», пример № 6) – для этого мы искали первый элемент с начала массива, больший x и первый элемент с конца массива, меньший x. После этого мы переставляли эти элементы местами и начинали искать следующую пару элементов. Когда станет ясно, что переставлять больше нечего, алгоритм закончит работу.

Теперь давайте продемонстрируем, как будет работать быстрая сортировка для следующего массива:

```
2    1    4    3    2    -1    8    1    5
```

Средний элемент – это M[5]=2.

После применения алгоритма перестановки элемента, получим следующий массив:

```
2    1    1    -1    2    |3    8    4    5
```

Теперь применяем алгоритм быстрой сортировки по очереди к первой и второй частям массива.

В первой части центральный элемент – это M[3]=1. Во втором – M[7]=8.

Первая часть массива после перестановки элементов примет вид

```
-1    1    1    |2    2
```

Теперь надо отсортировать две части этого куска массива и т.д.

Для того чтобы вы оценили всю мощь быстрой сортировки, мы проведем сравнение скоростей быстрой сортировки и сортировки пузырем. Интересно, как сильно будут отличаться результаты?

Пример 6: Быстрая сортировка.

```
1 : uses Crt,Dos;
2 : const
3 :   n=34000;
4 : type
5 :   Mas=array [1..n] of byte;
6 : var
7 :   A:Mas;
8 :   i:longint;
9 :   Uhr,Min,Sek,mSek:word;{часы, минуты, секунды, миллисек.}
10:   Uhr2,Min2,Sek2,mSek2:word;
11:
12: procedure BlaseSort(var M:array of byte);
13: var
14:   i,j:longint;
15:   tmp:integer; {Вспомогательная переменная}
16: begin
17:   for j:=High(M)-1 downto 1 do
18:     for i:=1 to j do
19:       if M[i]>M[i+1] then
20:         begin
21:         tmp:=M[i+1];
22:         M[i+1]:=M[i];
23:         M[i]:=tmp;
24:         end;
25: end;
26:
27: procedure SchnSort (var A:array of byte;L,R:longint);
28: var
29:   B,Tmp:integer;
30:   i,j:word;
31: begin
32:   B:=A[(L+R) div 2];
33:   i:=L;
34:   j:=R;
35:   while i<=j do{Делим массив на 2 части}
36:     begin
37:     while A[i]<B do
38:       i:=i+1;
39:     while A[j]>B do
40:       j:=j-1;
41:     if i<=j then
42:       begin
43:       Tmp:=A[i];
44:       A[i]:=a[j];
45:       A[j]:=Tmp;
46:       i:=i+1;
47:       j:=j-1;
48:       end;
49:     end;
50:   if L<j then SchnSort(A,L,j);  {Сортируем}
51:   if i<R then SchnSort (A,i,R); {обе части}
```

```
52: end;
53:
54: BEGIN
55:   ClrScr;
56:   randomize;
57:   for i:=1 to n do
58:     A[i]:=random(256);
59:   GetTime(Uhr,Min,Sek,mSek);{Время начала работы процедуры}
60:   BlaseSort(A);
61:   GetTime(Uhr2,Min2,Sek2,mSek2);{Время конца работы процедуры}
62:   writeln('Начало работы ',Uhr,':',Min,':',Sek,':',mSek);
63:   writeln('Конец работы ',Uhr2,':',Min2,':',Sek2,':',mSek2);
64:
65:   for i:=1 to n do  {повторно заполняем}
66:     A[i]:=random(256);
67:   GetTime(Uhr,Min,Sek,mSek);
68:   SchnSort(A,1,n);
69:   GetTime(Uhr2,Min2,Sek2,mSek2);
70:   writeln('Начало работы ',Uhr,':',Min,':',Sek,':',mSek);
71:   writeln('Конец работы ',Uhr2,':',Min2,':',Sek2,':',mSek2);
72:   readln;
73: end.
```

Для того чтобы узнать время начала и конца работы процедуры можно использовать находящиеся в модуле DOS процедуры GetTime. В нее надо передать часы, минуты, секунды и миллисекунды, и она запишет в них текущее время.

8.10. Генерирование перестановок

Пусть заданы некоторые числа a_1, a_2, …, a_n. Надо сгенерировать всевозможные перестановки этих чисел. Рассмотрим сначала основные понятия, которые могут понадобиться нам для решения задач.

1. Количество всевозможных перестановок n различных предметов $P_n = n!$

Например: перестановки трех чисел a, b, c : abc, acb, cab, bac, bca, cba.

2. Размещениями из n элементов по m называются такие их соединения, которые различаются друг от друга самими элементами или их порядком.

Пример: размещения элементов a, b, c по 2: ab, ba, ac, ca, bc, cb.

Число всех размещений из n различных элементов по m обозначается $A_n^m = \dfrac{n!}{(n-m)!}$

Очевидно, что $P_n = A_n^n$

3. Сочетаниями из n элементов по m называются их соединения, различающиеся друг от друга только самими элементами.

Пример: сочетания из a, b, c по 2: ab, ac, cb.

Число всех сочетаний из n различных элементов по m обозначается $C_n^m = \dfrac{A_n^m}{P_m} = \dfrac{n!}{m!(n-m)!}$

Все эти формулы элементарно доказываются:

1. Докажем, что $A_n^m = \frac{n!}{(n-m)!}$: первый элемент размещения выбирается n способами, второй - $(n-1)$-м способом, … m-й элемент можно выбрать $n-m+1$ способом.
   Значит $A_n^m = n\cdot(n-1)\cdot\ldots\cdot(n-m+1) = \frac{n!}{(n-m)!}$. Отсюда получаем и $P_n = A_n^n = n!$
2. Очевидно, что если некоторый набор является размещением, то и все его перестановки будут также являться размещениями. Так как все эти размещения образуют одно и то же сочетание, то выходит, что $C_n^m = \frac{A_n^m}{P_m} = \frac{n!}{m!(n-m)!}$.

Пример 7: Генерация всех перестановок (Umstellungen) некоторого множества.

```
1 :  uses Crt;
2 : const
3 :   n=3;
4 : type
5 :   Mas=array[1..n] of integer;
6 : var
7 :   A,B,C:Mas;
8 :
9 : procedure RandMass(var A:Mas;k:integer);
10: var
11:   i:integer;
12: begin
13:   for i:=1 to n do
14:     A[i]:=random(k);
15: end;
16:
17: procedure SchreibMass(var A:Mas);
18: var
19:   i:integer;
20: begin
21:   for i:=1 to n do
22:     write(A[i],' ');
23: end;
24:
25: { B[i]=0, если элемент A[i] уже вошел в перестановку
26:   В массиве C находятся по порядку индексы элементов
27: массива A, которые входят в текущую перестановку
28:   ind - номер элемента, который выбирается для
29:   включения в перестановку.
30: }
31: procedure Umstellungen(var A,B,C:Mas;ind:integer);
32: var
33:   i,j:integer;
34: begin
35:   for i:=1 to n do
36:     if B[i]=0 then{если нашли пустой элемент}
37:       begin
38:       C[ind]:=i;{заносим индекс в перестановку}
39:       if (ind=n) then{Если перестановка сформирована}
```

```
40:          begin
41:          for j:=1 to n do {печатаем элементы массива A}
42:            write(A[C[j]],' ');
43:          writeln;
44:          exit;
45:          end;
46:         B[i]:=1;{i-й элемент массива вошел в перестановку}
47:         {формируем следующий элемент перестановки}
48:         Umstellungen(A,B,C,ind+1);
49:         B[i]:=0;{Освобождаем i-й элемент}
50:         end;
51: end;
52:
53: begin
54:   Clrscr;
55:   randomize;
56:   RandMass(A,100);{Строим числовой массив}
57:   write('Mas A = ');
58:   SchreibMass(A);
59:   writeln;
60:   Umstellungen(A,B,C,1);
61:   readkey;
62: end.
```

8.11. Переборные алгоритмы

Часто встречаются задачи, которые надо решать с помощью перебора всех возможных вариантов. Но, как правило, полный перебор занимает очень много времени, поэтому надо искать способы уменьшения области перебора. Каких-то общих рекомендаций для решения переборных задач нет, поэтому необходим «индивидуальный подход к каждому клиенту».

Мы с вами рассмотрим классическую задачу: найти все расстановки n ферзей на доске $n \times n$, при которых ни один из ферзей не может побить другого.

Очевидно, что для $n = 1, 2, 3$ таких расстановок не существует. Для $n = 4$ задача имеет 2 решения.

Давайте для удобства горизонтали и вертикали нумеровать числами. Тогда задача сводится к нахождению перестановки $\begin{pmatrix} 1 & 2 & \dots & n \\ y_1 & y_2 & \dots & y_n \end{pmatrix}$, где каждая пара $(i \ \ y_i)$ соответствует координате некоторого ферзя.

Решение задачи сводится к одной рекурсивной функции StellNeueY. Она принимает в качестве параметров массив – 2-ю строку подстановки и номер столбца ind, на который надо поставить следующего ферзя. Ее алгоритм прост: просматриваем все поля ind-того столбца (цикл на строке 24), и смотрим, есть ли в массиве хотя бы один ферзь, который может побить ферзя, стоящего на поле $(ind \ \ j)$. Если такой ферзь есть, то проверяем следующее поле, если же таких ферзей нет, то ставим ферзя на это поле (строка 36) и вызываем процедуру StellNeueY для следующего поля. Если же ферзи уже поставлены на свои места, то печатаем перестановку.

Пример 8: Задача о 8 ферзях.

```
1 : uses
2 :   Crt;
3 : const
4 :   n=12;{Размер доски}
5 : type
6 :   MasY=array[1..n] of byte;
7 :
8 : procedure SchrMas(var M:MasY);{Вывод подстановки}
9 : var
10:   i:integer;
11: begin
12:   for i:=1 to n do
13:     write(i, ' ');
14:   writeln;
15:   for i:=1 to n do
16:     write(M[i],' ');
17:   writeln;
18: end;
19:
20: procedure StellNeueY(var M:MasY;ind:byte);
21: var
22:   j,k:byte;
23: begin
24:   for j:=1 to n do{просматриваем все поля i-го столбца}
25:     begin
26:     for k:=1 to ind-1 do{Проверяем, можно ли поставить ферзя на
j-е поле}
27:       if ((ind+j)=(k+M[k]))or((ind -j)=(k-M[k]))or(j=M[k]) then
28:         begin
29:         k:=ind;
30:         break;
31:         end;
32:     if (k=ind) then{Если поставить нельзя}
33:       continue   {Переходим на новое поле}
34:     else
35:       begin
36:       M[ind]:=j; {Ставим ферзя}
37:       if (ind=n) then{Если уже заполнили всю доску}
38:         begin
39:         SchrMas(M);
40:         writeln;
41:         end
42:       else   {Пытаемся поставить ферзя на следующие столбцы}
43:         StellNeueY(M,ind+1);
44:       end;
45:     end;
46: end;
47:
48: var
49:   M:MasY;
50: begin
```

```
51:   ClrScr;
52:   StellNeueY(M,1);
53:   readln;
54: end.
```

8.12. Косвенная рекурсия

Иногда может возникнуть следующая ситуация: подпрограмма A вызывает подпрограмму B, а та, в свою очередь, снова вызывает подпрограмму A. В таком случае мы имеем дело с косвенной рекурсией. Отличный пример косвенной рекурсии - стихотворение М. Ю. Лермонтова «Сон»[11]:

В полдневный жар в долине Дагестана
С свинцом в груди лежал недвижим я;
Глубокая еще дымилась рана,
По капле кровь точилася моя.

Лежал один я на песке долины;
Уступы скал теснилися кругом,
И солнце жгло их желтые вершины
И жгло меня – но спал я мертвым сном.

И снился мне сияющий огнями
Вечерний пир в родимой стороне.
Меж юных жен, увенчанных цветами,
Шел разговор веселый обо мне.

Но в разговор веселый не вступая,
Сидела там задумчиво одна,
И в грустный сон ее душа младая
Бог знает чем была погружена;

И снилась ей долина Дагестана;
Знакомый труп лежал в долине той;
В груди его, дымясь, чернела рана,
И кровь лилась хладеющей струей.

Из реальной жизни героя вызывается тема сна главного героя, в ней вызывается тема жизни любимой девушки, а в той – тема сна девушки, в которой снова вызывается тема жизни главного героя. Заметьте, что выхода из рекурсии нет, - дополнительное свидетельство смерти героя.

В TP с косвенной рекурсией возникает проблема: вы знаете, что TP последовательно просматривает программный код и если видит неизвестный идентификатор, моментально сообщает об ошибке. Значит, если подпрограмма A вызывает подпрограмму B, то эта подпрограмма B должна быть описана раньше, чем

---

[11] Этот стих в качестве примера косвенной рекурсии привел Анисимов А.В. в книге Информатика. Творчество. Рекурсия.

А, но она, в свою очередь, также вызывает подпрограмму А, а значит А должна быть описана раньше, чем В.

Рассмотрим пример, в котором показано, как можно выйти из затруднения. Будем вычислять значение функции $f(n) = a(n) + a([n/2]) + a([n/5])$, где

$$a(n) = \begin{cases} 2 \cdot f(n-1), & n \vdots 2, \quad n > 0 \\ 3 \cdot f(n-3), & n \not\vdots 2, \quad n > 0 \\ 1, & n \in \{0,1\} \end{cases}$$

Пример 9: Косвенная рекурсия.

```
1 : uses Crt;
2 : {Это объявление надо помещать в программу перед описанием
функции f}
3 : function a(n:longint):longint;forward;
4 :
5 : function f(n:longint):longint;
6 : begin
7 :   f:=a(n)+a(n div 2)+a(n div 5);
8 : end;
9 :
10: function a(n:longint):longint;
11: begin
12:   if (n=1) or (n=0) then
13:     a:=1
14:   else
15:     if (n mod 2 =0) then
16:       a:=2*f(n-1)
17:     else
18:       a:=3*f(n-3);
19: end;
20:
21: var
22:   i:integer;
23: begin
24:   Clrscr;
25:   for i:=1 to 15 do
26:     writeln(f(i));
27: end.
```

Вообще говоря, в этом примере можно обойтись и без косвенной рекурсии: просто подставить функцию $a(n)$ в функцию $f(n)$, и свести решение задачи к обычной рекурсии, но в таком случае количество строк кода значительно увеличится, так как условные операторы, находящиеся в функции a придется писать три раза. Для того, чтобы воспользоваться косвенной рекурсией, можно использовать дополнительное ключевое слово forward, которое дает понять компилятору, что само описание функции следует дальше; заголовок подпрограммы с директивой forward надо помещать перед первым упоминанием о функции.

Сразу хочется спросить: можно ли организовать косвенную рекурсию без слова forward?

Ответ: Да, можно. В следующем примере именно так и будет сделано.
Решать будем похожую задачу: будем вычислять $f\left(a(n)\right)=a(n)+a([n/2])+a([n/5])$, где $a(n)$ - некоторая функция, вид которой заранее не известен, но известно то, что она может выражаться через саму функцию $f\left(a(n)\right)$.
Заметьте: в предыдущем примере функция f зависела от переменной n, а сейчас – от функции $a(n)$. Значит, нам придется орудовать с функциональным типом.

Пример 10: Косвенная рекурсия без forward.

```
1 : uses Crt;
2 : type
3 :   Funk=function(n:longint;fun:Funk):longint;
4 :
5 : {Теперь мы передаем в функцию f не какую-то конкретную функцию,
6 : которую надо было описать выше, а ссылку на некоторую функцию}
7 : function f(n:longint;a:Funk):longint;
8 : begin
9 :   f:=a(n,a)+a(n div 2,a)+a(n div 5,a);
10: end;
11:
12: {Так как a(n) тоже вызывает f, то ссылку придется передавать
13: и в саму функцию a, т.е. одним из параметров функции должна быть
14: ссылка на саму же функцию}
15: function a(n:longint;fun:funk):longint;far;
16: begin
17:   if (n=1) or (n=0) then
18:     a:=1
19:   else
20:     if (n mod 2 =0) then
21:       a:=2*f(n-1,fun)
22:     else
23:       a:=3*f(n-3,fun);
24: end;
25:
26: var
27:   i:integer;
28: begin
29:   Clrscr;
30:   for i:=1 to 15 do
31:     writeln(f(i,a));
32: end.
```

Как видите, рекурсия без forward осуществима, причем мы решили более общую задачу, чем в предыдущем примере; но при этом пришлось перейти от функций к функционалам, что усложняет программный код, а также увеличивает количество параметров, которые надо передавать в функции. Естественно, теоретическая возможность обойтись без forward еще не означает, что пользоваться ею не стоит.

8.13. В чем итерация выигрывает у рекурсии

Вы знаете, что при вызове функции ПК запоминает адрес, откуда она была вызвана, и резервирует место под параметры вызова функции и локальные переменные. Также вы знаете, что при рекурсивных вызовах все эти параметры, локальные данные и адреса точек вызова должны складироваться в памяти. Кроме дополнительных затрат памяти добавляются также затраты ресурсов ПК на выделение этой памяти. Ясное дело, что если эти затраты относительно невелики и количество рекурсивных вызовов экспоненциально не растет, то ними можно пренебречь. Я же хочу обратить ваше внимание на следующий важный факт: **в рекурсии всегда есть обратный ход**. Когда я писал о сводимости итерационных алгоритмов к рекурсивным, то я рассуждал так: каждый новый виток цикла можно смоделировать просто рекурсивным вызовом подпрограммы с новыми параметрами. Но в итерации нет обратного хода: после перехода на новую итерацию цикл уже не возвращается к старой. Это зачастую и придает циклам бОльшую естественность. Например, хотя рекурсивная версия алгоритма Евклида и выглядит внешне красиво, но во время обратного хода надо постоянно передавать от одной копии подпрограммы к другой одно и то же число.

На мой взгляд, было бы неплохо добавить в язык программирования возможность рекурсивного вызова подпрограммы без необходимости рекурсивного возврата. Это можно было бы сделать так: когда некоторая подпрограмма A вызывает рекурсивную подпрограмму B, запоминать место возврата, а когда сама процедура B начинает вызывать свои копии, то стирать все параметры и локальные данные, принадлежащие материнской функции B, а адресом возврата дочерней функции B сделать адрес возврата материнской функции B. При такой реализации рекурсии на каждом новом шаге рекурсии будут храниться лишь параметры и локальные данные для последней копии процедуры B а также адрес ее возврата в функцию A.

Естественно, что при таком методе на структуру подпрограммы B надо наложить определенные ограничения, а именно:

1. В рекурсивной подпрограмме может быть только 1 вызов дочерней функции (возврата не будет, поэтому 2-ой вызов был бы просто невозможен.
2. Т.к. возврата нет, то все действия должны совершаться на рекурсивном спуске.

Ясно, что для итерационных алгоритмов эти условия выполняются, поэтому такая модификация рекурсии для них идеально подходит.

Возможность такой «модернизированной» рекурсии есть, например, в Prolog, где она называется хвостовой рекурсией.

8.14. Рекурсия в LISP

В функциональных языках программирования, таких как LISP, основным понятием является функция. Вообще говоря, в LISP есть и итерационные структуры, и условия, но все они записываются в виде функций.

Например, алгоритм Евклида выглядит на LISP так:

Пример 11: Алгоритм Евклида на LISP.

```
(defun Eukl (a b)      ; объявление функции Eukl(a,b)
  (if (= b 0)          ;if b=0
    a                  ;вернуть в качестве результата b
    (Eukl b (mod a b));вернуть в качестве результата Eukl(b,a mod b)
```

```
  )                     ;конец if
)                       ;конец описания функции
```

Обратите внимание: для того, чтобы использовать функцию в императивных языках, надо использовать оператор вызова функции (т.е. базовым понятием является все-таки оператор). А в LISP операции и операторы выглядят как функции (т.е. базовым понятием является функция). В этом и различие между функциональными и императивными языками.

Заметьте, что в LISP нет специального оператора возвращения результата функции. Одним из следствий этого является то, что можно избавиться от оператора присваивания: передавать данные только как аргументы функций, а полученные результаты сразу передавать как параметры другим функциям.

8.15. Сопрограммы

Косвенная рекурсия приводит к еще одному очень важному следствию. Ранее мы говорили о локальных и глобальных переменных и подпрограммах, и отметили тот факт, что понятия «локальный» и «глобальный» относительны. Обыкновенная рекурсия привела к тому, что копии одной и той же функции могут быть вложены одна в другую и одна копия функции может быть глобальной по отношению к другой копии; но по отношению ко всем остальным подпрограммам, которые вызывают рекурсивную функцию, сама она остается локальной, и если считать рекурсивную функцию со всеми ее копиями единым целым, то понятия «локальный» и «глобальный» сохраняют свою суть. С косвенной рекурсией дело обстоит еще интереснее: когда одна подпрограмма вызывает другую, а та, в свою очередь, третью, которая может вызвать снова первую,…, - то считать косвенно-рекурсивную подпрограмму чем-то отдельным мы не можем, более того, - если раньше могло показаться, что понятие «локальный» означает также и «подчиненный», то в случае косвенной рекурсии понятия «главный» и «подчиненный» совсем теряют смысл. Конечно, мы должны помнить, что для каждой копии подпрограммы компилятор создает новые переменные под локальные данные, поэтому копии одной и той же функции владеют разными локальными данными.

Можно пойти еще дальше, и ввести понятие сопрограмм.

- 2 подпрограммы назовем **сопрограммами**, если каждая из них может приостанавливать свое выполнение, и возобновлять выполнение другой подпрограммы (не копии, а **той же самой** подпрограммы).

На рис. 8.4 показан пример двух сопрограмм: сначала начинает выполняться первая сопрограмма, затем, выполнив 2 оператора, она вызывает 2-ю сопрограмму, которая выполняет 3 оператора и возобновляет работу первой (а не вызывает новую копию 1-й сопрограммы). Потом 1-я сопрограмма начинает выполнять свои операторы и затем возобновляет работу 2-й сопрограммы и т.д.

Несмотря на то, что в каждый момент времени выполняется лишь одна из сопрограмм, тем не менее с их помощью можно описывать параллельные процессы.

В TP нет встроенных средств, позволяющих описывать сопрограммы. Но когда мы будем изучать Delphi, мы рассмотрим иной механизм, который позволит моделировать параллельные процессы. О сопрограммах и о языке Simula, который поддерживает их, мы еще вспомним в последней главе этой книги.

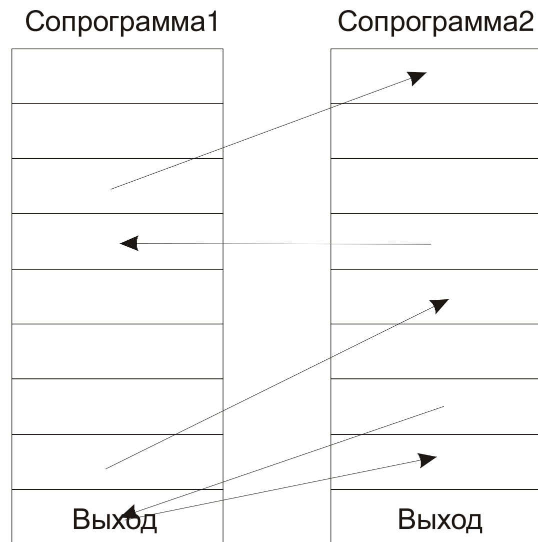


Рис 8.4. Работа сопрограмм

Задачи

1. Вычислите периметр и площадь острова Коха.
2. Напишите рекурсивную функцию вычисления $n!$, основываюсь на следующем определении: 0!=1, $n! = n \cdot (n-1)!$, $n > 0$.
3. Вычислите с помощью рекурсии $\cos(\cos(...(\cos(x)))$, где глубина вложенности косинуса = n.
4. Докажите формулы $C_n^0 = C_n^n = 1$, $C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$
5. Придумайте алгоритм вычисления a^n, где n - натуральное, работающий за время $c\log_2 n$.
6. Придумайте итерационный алгоритм решения задачи о Ханойских башнях.
7. Подсчитайте для задачи о ханойских башнях количество перемещений дисков, которое необходимо для переноса пирамиды из n дисков.
8. В задаче о Ханойских башнях диски можно перемещать лишь одним способом (можно, конечно, каждый диск перекладывать с места на место по несколько раз, но этот случай мы не рассматриваем). Задача значительно усложняется, если количество стержней больше 3-х. Попытайтесь найти оптимальный алгоритм и количество перемещений дисков, если количество столбцов = 4, а затем обобщите задачу на случай переноса n дисков с помощью m стержней.
9. Можно ли написать рекурсивную подпрограмму, которая в процессе выполнения *может вызвать* саму себя с теми же входными параметрами, но при этом бесконечная рекурсия все равно может не возникнуть? Обоснуйте ответ.
10. Можно ли написать рекурсивную подпрограмму, которая во время работы *обязательно* вызовет себя хотя бы один раз с теми же параметрами, но при этом бесконечной последовательности рекурсивных вызовов не будет?
11. Напишите рекурсивную подпрограмму, которая ищет в массиве элемент методом бинарного поиска.
12. Заданное натуральное число необходимо разбить на всевозможные суммы натуральных чисел без повторений, с точностью до перестановки слагаемых. Результаты вывести на экран.

13. Написать программу определения количества шестизначных счастливых билетов (билетов, у которых сумма первых трех цифр равна сумме трех последних).
14. Модернизируйте программу о n ферзях так, чтобы она выдавала количество расстановок.
15. Найдите минимальное количество ферзей, которые надо поставить на шахматную доску размера $n \times n$, чтобы при этом они держали под боем все поля доски
16. Написать программу обхода шахматной доски конем так, чтобы он на каждом поле побывал по одному разу.
17. Вам попалась карта острова в виде матрицы $n \times m$, заполненной нулями и единицами (1-суша, 0 – море). Остров – это совокупность элементов суши, граничащих между собой по горизонтали или вертикали. Надо вывести на экран размер острова.
18. Пусть у вас есть карта целого архипелага, и количество островов в нем. Надо найти размер максимального острова.
19. Пусть у вас есть процедура, которая упорядочивает массив методом быстрой сортировки (в которой выбирается центральный элемент для разбиения массива на 2 части). Напишите программу, которая по числу n строит массив, который быстрая сортировка может упорядочить только за время cn^2. Сравните скорость сортировок пузырьком и быстрой сортировки для таких массивов.

Глава 9: Модули и записи

Когда количество строк кода в наших программах превысило 100, нам пришлось разбивать нашу программу на более мелкие части - подпрограммы. Но обычно само количество подпрограмм в проекте больше 100, поэтому мы приходим к той же проблеме, лишь на новом уровне. Но что выясняется: TP позволяет разбивать программу не только на отдельные подпрограммы, но и на отдельные файлы (которые называются модулями), каждый из которых является целой коллекцией подпрограмм.

9.1. Модули и их структура

- Модуль – это отдельно компилируемая программная единица, содержащая набор типов, подпрограмм, переменных, констант, …, которые могут экспортироваться в основную программу.

Модули подключаются к программе с помощью директивы Uses (кроме модуля System, который подключается автоматически).

Структура модулей:

```
Unit имя;

Interface
интерфейсная часть

Implementation
исполняемая часть

BEGIN
инициализирующая часть
END.
```

Ключевое слово unit означает, что данный файл – модуль. Название модуля должно совпадать с именем файла с исходным кодом. Например, если:
unit Cmplx;
то название файла, в котором написан модуль, должно быть cmplx.pas.

Интерфейсная часть

Начинается со слова interface; содержит объявления всех глобальных типов, констант и переменных, которые должны быть доступны в той программе (или другом модуле), которая подключает данный модуль. При объявлении глобальных подпрограмм в интерфейсной части указывается только их заголовок.

Исполняемая часть

Начинается словом implementation и содержит реализации подпрограмм, объявленных в интерфейсной части. В ней могут объявляться локальные для модуля объекты: вспомогательные типы, константы, переменные.

Когда вы пишете подпрограмму, объявленную в интерфейсной части модуля, вы можете писать только название процедуры или функции, опуская список формальных параметров и тип результата для функции. Но так делать не очень удобно, так как

обычно сначала пишется реализация функции, а затем ее заголовок копируется в интерфейсную часть, а не наоборот.

Кроме того, в части implementation можно писать подпрограммы, не объявленные в части interface. В таком случае эти подпрограммы могут быть использованы лишь в других подпрограммах этого модуля, а вне его они не видны. Так рекомендуется делать с теми подпрограммами, которые крайне узкоспециализированы, и вне модуля они не пригодятся.

Инициализирующая часть

Начинается словом Begin. В этой части располагаются исполняемые операторы, содержащие какую-то часть программы. Обычно это некоторые вспомогательные действия: инициализация графического режима, открытие или создание нужных файлов. Все действия инициализирующей части выполняются до передачи управления основной программе.

Инициализирующая часть может отсутствовать. В таком случае слово begin не пишется, и после части implementation следует сразу END.

Компиляция модулей

Модуль не может запускаться на выполнение самостоятельно. Он может участвовать только при построении программы или другого модуля. Для того, чтобы вы могли использовать модуль, надо предварительно откомпилировать его. В результате компиляции создается tpu-файл (Turbo Pascal Unit). Файлы с расширением tpu содержат исходный код модуля в специальном формате, поэтому прочитать их нельзя (точнее: прочитать можно, но понять нельзя). При компиляции основной программы эти файлы преобразуются в машинный язык. Конечно, можно было бы обойтись без промежуточных файлов. Но тогда было бы больше проблем с защитой авторских прав, т.к. если вы пишете не программу, а модуль, то для того, чтобы он мог быть использован в другой программе, вы должны предоставить его исходный код. Если же есть промежуточные .tpu-файлы, то исходный код никто, кроме вас знать не будет. Не менее важно то, что если вы пишете большой проект, то его компиляция занимает много времени, а использование заранее скомпилированных модулей значительно снижает затраты времени.

В TP определены 3 режима компиляции:

COMPILE – при компиляции модуля или основной программы для всех модулей, которые указываются в директиве uses, должны существовать соответствующие tpu-файлы.

MAKE – компилятор проверяет для каждого объявленного в разделе uses модуля наличие .tpu файлов. Если tpu-файл есть, то для компиляции программы используется он. В противном случае компилятор проверяет, есть ли файл с исходным кодом. Если он есть, то этот файл компилируется в tpu-файл, а затем tpu-файл используется для компиляции программы. Причем если в исходный код модулей вносились изменения, то независимо от того, существует ли уже .tpu файл, или нет, программа откомпилирует модуль заново.

BUILD – существующие tpu-файлы не принимаются во внимание, и программа пытается найти и откомпилировать соответствующие pas-файлы для каждого объявленного в uses модуля.

Файл с расширением .tpu при сохранении модуля автоматически не создается: надо, чтобы этот модуль был откомпилирован какой-то программой в любом из приведенных режимов. Программа видит модуль, только если файл с исходным кодом (или соответствующий .tpu файл) находится в директории, которая указана в пункте меню Options -> Directories ->Unit Directories. Если там записана пустая строка, то это означает, что модули должны находиться в том же каталоге, что и turbo.exe.

Итак, чтобы создать модуль вы должны:

1. Создать pas-файл с исходным кодом
2. Подключить его к какой-то программе, откомпилировать его в любом режиме и получить tpu-файл.

Пример 1: Модуль для работы с массивами.

```
1 : unit Massiv;
2 :
3 : Interface
4 :
5 : procedure LeseMas(var A:array of integer);
6 : procedure SchrMas(const A:array of integer);
7 : function BinSuche(const A:array of integer;left,right:integer;
numb:integer):longint;
8 : procedure AuswahlSort(var A:array of integer);  {Auswahl -
выбор}
9 : procedure RandFulle(var A:array of integer;k:integer);
10:
11: Implementation
12:
13: procedure LeseMas(var A:array of integer);
14: var
15:   i:longint;
16: begin
17:   for i:=0 to High(A) do
18:     read(A[i]);
19: end;
20:
21: procedure SchrMas(const A:array of integer);
22: var
23:   i:longint;
24: begin
25:   for i:=0 to High(A) do
26:     write(A[i],' ');
27: end;
28:
29: function BinSuche(const A:array of integer;left,right:integer;
numb:integer) :longint;
30: var
31:   mid:longint;
32: begin
33:   mid:=round((left+right)/2);
34:   if A[mid]=numb then
35:     begin
```

```
36:     BinSuche:=mid;
37:     exit;
38:     end;
39:   if left>=right then
40:     begin
41:     BinSuche:=0;
42:     exit;
43:     end;
44:   if A[mid]>numb then
45:     BinSuche:=BinSuche(A,left,mid-1,numb)
46:   else
47:     BinSuche:=BinSuche(A,mid+1,right,numb);
48: end;
49:
50: procedure AuswahlSort(var A:array of integer);
51: var
52:   i,j,s:longint;
53:   min:integer;
54: begin
55:   for i:=0 to High(A)-1 do
56:     begin
57:     min:=A[i];
58:     s:=i;         {Индекс минимального элемента}
59:     for j:=i+1 to High(A) do
60:       if A[j]<min then {Если нашелся элемент, меньший данного}
61:         begin          {присваиваем его переменной min}
62:         min:=A[j];
63:         s:=j;        {В s запишем индекс нового мин. элемента}
64:         end;
65:     A[s]:=A[i];           {Меняем A[i] и min местами}
66:     A[i]:=min;
67:     end;
68: end;
69:
70: procedure RandFulle(var A:array of integer;k:integer);
71: var
72:   i:longint;
73: begin
74:   for i:=0 to High(A) do
75:     A[i]:=random(k);
76: end;
77:
78:
79: END.
```

Использование модуля:

```
1 : uses
2 :   Massiv;
3 : const
4 :   n=100;
5 : type
6 :   Mas = array[1..n] of integer;
```

```
7 : var
8 :   A:Mas;
9 :
10: begin
11:   randomize;
12:   RandFulle(A,1000);
13:   writeln('Исходный массив');
14:   SchrMas(A);
15:   writeln;
16:   AuswahlSort(A);
17:   writeln('Отсортированный массив');
18:   SchrMas(A);
19:   writeln;
20: end.
```

9.2. Взаимодействие модулей

При разработке программных комплексов с большим количеством модулей могут появляться проблемы согласования их работы. Основной проблемой является зацикливание, т.е. случай, когда модуль A подключается в модуле B, а модуль B - в модуле A. Эта проблема возникает настолько часто, что знать методы ее решения надо обязательно!

Следующий программный код неверен:

```
Unit A;
Interface
Uses B;
…
Implementation
…
End.

Unit B;
Interface
Uses A;
…
Implementation
…
End.
```

Компилятор TP подключает модули последовательно, поэтому в этом фрагменте программы прежде, чем подключить модуль A надо подключить модуль B, и наоборот: перед подключением модуля B надо подключить A. Поэтому компилятор так написать не позволит.

Если хотя бы один из двух модулей зависит от другого только в части implementation, то тогда программу можно исправить так:

```
Unit A;
Interface
```

```
…
Implementation
Uses B;
…
End.

Unit B;
Interface
Uses A;
…
Implementation
…
End.
```

Теперь TP не выдаст ошибки, и программа будет откомпилирована нормально. Если же взаимозависимы части interface, то решить проблему сложнее.

Очень часто отделаться удается малой кровью: просто создать 3-й модуль C, в который поместить все типы данных, подпрограммы и переменные, которые ссылаются друг на друга в модулях A и B. После этого убрать их из обоих модулей, а модуль C подключить и в A, и в B.

К сожалению, это спасает не всегда. Могут существовать проблемы, когда даже внутри одного модуля нельзя описать 2 типа из-за того, что они ссылаются друг на друга. Однако такие случаи появятся, когда мы начнем использовать классы, поэтому их рассмотрение отложим до главы «ООП», где сразу и решим эту проблему.

9.3. Тип Запись

- Запись – объединение нескольких переменных в единое целое.
- Элементы записи называются полями.

Поля записи могут быть любого типа, доступного в TP (в том числе и другие записи). Одним записям можно присваивать другие записи того же типа.
Синтаксис описания записи:

имя типа = record
    список полей;
end;

Простейший пример использования записи – работа с рациональными числами. Рациональное число представляется в виде двух целых чисел – числителя и знаменателя. Поэтому запись дробь будет выглядеть так:

```
type
  Bruch= record  {Объявление типа Bruch (дробь)}
    Zahler:longint; { Zähler = числитель}
    Nenner:longint;{знаменатель}
  end;
```

Переменные созданного нами типа объявляются как обычно, например:

```
Var
  A:Bruch;
```

Доступ к полям записи можно получить с помощью операции ‘.’. Вначале нужно ставить название переменной, затем точку, далее – название поля, к которому надо получить доступ.

В следующих строках числителю дроби A присваивается значение 14, а знаменателю – 15.

```
A.Zahler:=14;
A.Nenner:=15;
```

Передавать переменные типа запись в подпрограммы будем так же, как мы это делали с массивами: как параметр-переменную, либо как параметр-константу. По значению передавать записи не желательно, т.к. при этом неэффективно расходуется память.

В качестве примера мы напишем модуль, содержащий подпрограммы для работы с дробями.

Для записей не подходят процедуры write и read, поэтому надо написать собственные подпрограммы ввода и вывода дробей на экран. Кроме того, надо реализовать арифметические и логические (> < = <>) операции над дробями. Все эти подпрограммы должны лежать в части interface, чтобы они были доступны для пользователя модуля.

Кроме того, модуль обязательно должен содержать подпрограммы сокращения дробей, проверки на правильность ввода (не равен ли знаменатель нулю), а также процедуру переноса знака «-» со знаменателя на числитель (см. упражнение №2). Эти подпрограммы не должны быть доступны для пользователя модуля (все эти операции должны проводиться автоматически, например, сокращать дроби надо после всех арифметических действий с дробями).

Пример 2: Модуль для работы с рациональными числами (в модуле Math.pas есть процедура вычисления НОД).

```
1 : unit BruchU;
2 :
3 : interface
4 :
5 : uses
6 :   Math; {нам нужна процедура GGT(a,b) для нахождения НОД}
7 : type
8 :   Bruch= record  {Объявление типа Bruch (дробь)}
9 :     Zahler:longint; {числитель}
10:     Nenner:longint;{знаменатель}
11:   end;
12:
13: procedure AddBruch(const Rat,Rat2:Bruch;var Rat3:Bruch);
14: {складывает числа Rat и Rat2 и записывает результат в Rat3}
15:
16: procedure MultBruch(const Rat,Rat2:Bruch;var Rat3:Bruch);
17: {умножает числа Rat и Rat2 и записывает результат в Rat3}
18:
19: procedure LeseBruch(var Rat:Bruch);{Ввод дроби}
20: procedure SchrBruch(var Rat:Bruch);{Печать Rat}
21:
22: function Grosser(const Rat1,Rat2:Bruch):boolean;
23: {Возвращает true, если Rat1>Rat2}
```

```
24:
25: implementation
26:
27: function Grosser(const Rat1,Rat2:Bruch):boolean;
28: begin {(a/b) > (c/d) <=> a*d>b*c }
29:   Grosser:=Rat1.Zahler*Rat2.Nenner>Rat2.Zahler*Rat1.Nenner;
30: end;
31:
32: procedure Kurzung(var Rat:Bruch);{сокращение дроби}
33: var
34:   c:longint;
35: begin
36:   c:=GGT(Rat.Zahler,Rat.Nenner);
37:   Rat.Zahler:=Rat.Zahler div c;
38:   Rat.Nenner:=Rat.Nenner div c;
39: end;
40:
41: procedure PrufRat(var Rat:Bruch);
42: {проверяет введенное число на правильность:
43: если Nenner=0, то присваивает дроби значение 1,
44: и печатает об этом пользователю}
45:
46: begin
47:   if Rat.Nenner = 0 then
48:     begin
49:     writeln('Знаменатель дроби = 0');
50:     Rat.Nenner:=1;
51:     Rat.Zahler:=1;
52:     writeln('Значение дроби будет установлено равным 1');
53:     end;
54: end;
55:
56: procedure LeseBruch(var Rat:Bruch);
57: begin
58:   readln(Rat.Zahler);  {вводим числитель дроби Rat}
59:   readln(Rat.Nenner);{вводим знаменатель дроби Rat}
60:   PrufRat(Rat);{проверяем, не равен ли знаменатель 0}
61:   Kurzung(Rat);{сокращение дроби}
62: end;
63:
64: procedure SchrBruch(var Rat:Bruch);
65:   begin
66:     if Rat.Nenner=1 then {a/1 = a  => write(a)}
67:       begin
68:       writeln(Rat.Zahler);
69:       exit;
70:       end;
71:     if Rat.Zahler=0 then  {0/d = 0  => write(0)}
72:       begin
73:       writeln(0);
74:       exit;
75:       end;
```

```
76:     writeln(Rat.Zahler,'/',Rat.Nenner);
77:   end;
78:
79: procedure AddBruch(const Rat,Rat2:Bruch;var Rat3:Bruch);
80:   begin   {(a/b) + (c/d) = (a*d+b*c)/(b*d)}
81:     Rat3.Nenner:=Rat.Nenner*Rat2.Nenner;
82:     Rat3.Zahler:=Rat.Zahler*Rat2.Nenner+Rat2.Zahler*Rat.Nenner;
83:     Kurzung(Rat3);{сокращение дроби}
84:   end;
85:
86: procedure MultBruch(const Rat,Rat2:Bruch;var Rat3:Bruch);
87:   begin  {(a/b) * (c/d) = (a*c)/(b*d)}
88:     Rat3.Zahler:=Rat.Zahler*Rat2.Zahler;
89:     Rat3.Nenner:=Rat.Nenner*Rat2.Nenner;
90:     Kurzung(Rat3);{сокращение дроби}
91:   end;
92: end.
```

//////////////// Основной файл:///////////////

```
Uses CRT,BruchU;
var
    A,B,C:Bruch;
BEGIN
  Clrscr;
  LeseBruch(A);
  LeseBruch(B);

  MultBruch(A,B,C);
  SchrBruch(C);
end.
```

9.4. Оператор with

Если в записи много полей, то часто бывает удобен оператор with
Например:

```
With A do
  feld1:=1;
  feld2:=5;
  feld3:='Ja';
end;
```

Эти строки эквивалентны следующим:

```
  A.feld1:=1;
  A.feld2:=5;
  A.feld3:='Ja';
```

9.5. Записи с вариантами

В TP есть еще один тип записей – записи с вариантами. Давайте рассмотрим его на примере:

Пример 3: Записи с вариантами.

```
1 : type
```

```
2 :   Student=record
3 :     {фиксированная часть}
4 :     name:string;
5 :     {вариантная часть}
6 :     case n:byte of
7 :     1:(Analysis1,Algebra,Diskrmath:byte);
8 :     2:(Progr,Analysis2,Geom,Geschichte:shortint);
9 :   end;
10: var
11:   s:student;
12: begin
13:   s.Analysis1:=5;
14:   s.Algebra:=5;
15:   s.Progr:=4; {Теперь и s.Analysis1=4}
16:   writeln(s.Analysis1,' ',s.Progr,' ',s.Algebra);
17:   readln;
18: end.
```

Описание типа запись в общем случае состоит из двух частей: фиксированной и вариантной. До этого мы имели дело лишь с фиксированной частью. Вариантная часть начинается конструкцией, внешне сходной на оператор выбора case, но это сходство лишь внешнее. Смысл вариантной части таков: одно и то же место в памяти ПК резервируется под 2 набора переменных. В примере длина максимального набора = 4 байта, минимального = 3 байта. Это означает, что под вариантную часть резервируется 4 байта (и еще 1 байт для селектора – переменной n). причем к первым 3-м байтам из 4-х можно обращаться под двумя именами (например, к первому байту – с помощью имени Analysis1 или имени Progr). Разумеется, т.к. оба набора хранятся в одной области памяти, то изменяя значения переменных в одном наборе вы автоматически измените и значения в другом наборе.
В данном примере на экран будет выведено: 4, 4, 5.

Задачи

1. Составить модуль с сортировками. Включить в него сортировки пузырем, вставками, выбором, быструю сортировку и др.
2. Пополните модуль работы с рациональными числами следующими подпрограммами:
   - Проверка правильности ввода дроби (не равен ли знаменатель нулю).
   - Деление двух дробей
   - Сокращение дроби. Эта процедура нужна не только для эстетической красоты, без нее числитель или знаменатель дроби может быстро стать больше чем максимальное число, которое может быть записано в переменной типа longint.
   - Смена знака (при работе с дробями может случиться, что в знаменателе окажется отрицательное число, а у нас по определению в знаменателе должно стоять натуральное число).
   - Сравнение дробей (<, =, <>).

3. **(!)** Напишите модуль, содержащий подпрограммы для работы с комплексными числами:
   - Ввод, вывод, сравнение с нулем, вычисление модуля комплексного числа
   - Перевод в тригонометрическую и показательную формы
   - Сложение, вычитание, умножение, деление, возведение в степень.
   - Решение квадратного уравнения с комплексными коэффициентами над множеством комплексных чисел.
4. Реализовать модуль для работы с многочленами на основе массивов
   - Сложение
   - Вычитание
   - Умножение
   - Деление
   - Возведение в степень (оптимальный алгоритм – см. задачу 5 в главе «Рекурсия»)

Глава 10: Матрицы, множества и перечисления

В 10-й и 11-й главах мы изучим 4 встроенных типа данных. Все они реализованы на основе массива, поэтому их изучение не даст нам новых фундаментальных понятий. Но эти типы данных используются довольно часто, поэтому желательно знать их.

10.1. Матрицы

Мы с вами уже научились работать с одномерными массивами. Теперь наша цель научиться работать с двумерными массивами, или матрицами. Трехмерные массивы используются крайне редко, поэтому рассматривать их не имеет смысла, т.к. принципы работы с ними те же.

- Матрица – двумерный массив.

Описывается прямоугольная целочисленная матрица размера n*m так:

```
type
  Matrix=array [1..n,1..m] of integer;
```

A[1,1]	A[1,2]	...	A[1,m]
A[2,1]	A[2,2]	...	A[2,m]
...	...	...	...
A[n,1]	A[n,2]	...	A[n,m]

Рис 10.1. Структура матрицы A типа Matrix

При этом в матрице будет n столбцов и m строк (см. рисунок). Матрицу можно описать иначе (как массив массивов):

```
type
  Matrix2=array [1..n] of array[1..m] of integer;
```

Вы можете описывать матрицу обоими этими способами, но при этом компилятор считает эти описания разными. Поэтому если вы захотите присвоить переменную типа Matrix переменной типа Matrix2, то без приведения типов не обойтись.

Диапазоны изменения элементов могут быть другие. Например:

```
type
  Mat=array['a'..'r',-3..14] of real;
```

Но такими экстравагантными индексациями мы пользоваться не будем.

Ограничения на размер переменной для матриц те же, что и на размер массива. Матрицы одного типа можно присваивать друг другу.

Обратиться к элементу матрицы можно, указав оба его индекса:

A[3,4]:=6

Вы знаете, что память в компьютере представляется в виде цепочки идущих друг за другом байтов; следовательно матрица, несмотря на то, что мы представляем ее себе как двумерный массив, должна храниться в памяти в виде строки. Хранится матрица в память компьютера так: сначала хранится первая строка матрицы, затем – вторая, и т.д. до последней строки. В принципе, то, каким образом матрица представляется в памяти, относится не к самому языку ТР, а к его компилятору. Поэтому в принципе, другие компиляторы могут иначе представлять матрицы в памяти ПК, например, постолбцово.

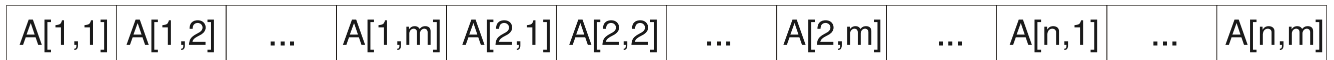


Рис 10.2. Так хранится матрица в памяти

Пример 1: Процедуры для ввода, вывода и заполнения матриц числами в определенном порядке.

```
1 : const
2 :   n=3; {количество строк матрицы}
3 :   m=2; {количество столбцов матрицы}
4 : type
5 : {Объявление вещественной матрицы размера n*m}
6 :   Matrix=array[1..n,1..m] of real;
7 : var
8 :   A:Matrix;
9 :
10: {Ввод прямоугольной вещественной матрицы}
11: procedure LeseMatr(var A:Matrix);
12: var
13:   i,j:integer;
14: begin
15:   for i:=1 to n do  {проход по строчкам матрицы}
16:     for j:=1 to m do {проход по столбцам матрицы}
17:       read(A[i,j]);
18: end;
19:
20: {Вывод прямоугольной вещественной матрицы на экран}
21: procedure SchrMatr(const A:Matrix);
22: var
23:   i,j:integer;
24: begin
25:   for i:=1 to n do
26:   begin
27:     writeln;{Переход на новую строку}
28:     for j:=1 to m do
29:       write(A[i,j]:9:4);
30:   end;
31: end;
32:
33: procedure EinfFulleMatr(var A:Matrix);
34: var
35:   i,j:integer;
36: begin
```

```
37:   for i:=1 to n do
38:     for j:=1 to m do
39:       A[i,j]:=i; {Заполняем всю i-ю строку числом i}
40: end;
41:
42: BEGIN
43:   writeln('Введите матрицу A размера ',n,' на ',m);
44:   LeseMatr(A);
45:   writeln('Матрица имеет такой вид A');
46:   SchrMatr(A);
47:   writeln;
48:
49:   writeln('Матрица, у которой элементы i-й строки =i');
50:   EinfFulleMatr(A);
51:   SchrMatr(A);
52:   readln;
53: end.
```

Вы видите, что принцип заполнения матриц прост: во внешнем цикле изменяется первый индекс (строчный), а во вложенный цикл пробегает по всей i-й строке от начала до конца. Выходит, что сначала заполняется вся первая строка (i=1), затем 2-я, и т.д.

10.2. Обобщение процедур работы с матрицами

Следующая наша цель – попытаться написать модуль для работы с матрицами любого размера. Для решения такой задачи для массивов мы использовали либо открытые параметры-массивы, либо бестиповые ссылки. Открытые массивы нам не подойдут, т.к. в качестве параметра передать в подпрограмму
A: array of array of integer
нельзя, т.к. array of integer – не тип данных.

Значит, надо использовать бестиповые ссылки.

Пример 2: Использование бестиповых ссылок.

```
1 : unit b2MatrU;
2 :
3 : interface
4 :
5 : type
6 :   GrosseMatr=array[1..100,1..100] of integer;
7 :
8 : procedure LeseMatr(var A;n,m:integer);
9 : procedure SchrMatr(var A;n,m:integer);
10:
11: implementation
12:
13: procedure LeseMatr(var A;n,m:integer);
14: var
15:   i,j:integer;
16: begin
17:   for i:=1 to n do
```

```
18:     for j:=1 to m do
19:       read(GrosseMatr(A)[i][j]);
20: end;
21:
22: procedure SchrMatr(var A;n,m:integer);
23: var
24:   i,j:integer;
25: begin
26:   for i:=1 to n do
27:     begin
28:     for j:=1 to m do
29:       write(GrosseMatr(A)[i][j],' ');
30:     writeln;
31:     end;
32: end;
33:
34: end.
```

А вот и файл, в котором этот модуль используется:

```
1 : uses b2MatrU;
2 : const
3 :   n=3; {количество строк матрицы}
4 :   m=2; {количество столбцов матрицы}
5 : type
6 : {Объявление вещественной матрицы размера n*m}
7 :   Matrix=array[1..n,1..m] of real;
8 : var
9 :   A:Matrix;
10:
11: begin
12:   writeln('Введите матрицу A размера ',n,' на ',m);
13:   LeseMatr(A,n,m);
14:   writeln('Матрица имеет такой вид A');
15:   SchrMatr(A,n,m);
16:   writeln;
17: end.
```

Для тех, кто уже подзабыл принцип работы с бестиповыми ссылками, напомню, что после передачи бестиповой ссылки в качестве параметра, мы должны привести эту ссылку к какому-то типу. В данном случае – к типу GrosseMatr (строки 19, 29). Он объявлен как квадратная матрица, но ясно, что все процедуры модуля будут отлично работать для любой подматрицы для матрицы 100*100 элементов. Вам все понятно? – Тогда попытайтесь распечатать матрицу A обычным способом, т.е. вставьте в основную программу вместо SchrMat(A,n,m) небольшой фрагмент:

```
  for i:=1 to n do
    begin
    for j:=1 to m do
      write(A[i,j],' ');
    writeln;
    end;
```

Вы не поверите своим глазам: правильно заполненной оказалась лишь 1-я строка, хотя, казалось бы, мы все делали правильно. Все дело вот в чем: вспомните, как хранится матрица размера 3×2 в памяти:

A[1,1]	A[1,2]	A[2,1]	A[2,2]	A[3,1]	A[3,2]

А матрица размеров 100×100 хранится так:

A[1,1]	A[1,2]	...	A[1,100]	A[2,1]	A[2,2]	...	A[2,100]	...	A[100,100]

В процедуре InputMatr мы заполняем элементы A[1,1], A[1,2], A[2,1], …, A[3,2], интерпретируя матрицу A как матрицу типа MegaMatr. Но ведь на самом деле A – матрица размера 3×2, поэтому для нее элемент, например, A[2,1] находится в памяти на совсем другом месте, чем элемент A[2,1], если матрица A – матрица размера 100×100.

Если вы напишете в основном блоке writeln(A[51,1]), то тогда вы увидите тот элемент, который вы ввели как элемент A[2,1] внутри процедуры InputMatr.

Уфф, какой я обманщик!!!

Теперь, после выявления ошибки, мы напишем еще одну программу, которая удовлетворит нашим требованиям.

Пример 3: Настоящее обобщение.

```
1 : unit MatrArrU;
2 :
3 : interface
4 :
5 : type
6 :   GrosseMass=array[1..MaxInt] of integer;
7 :
8 : procedure LeseMass(var A;n,m:integer);
9 : procedure SchrMass(var A;n,m:integer);
10: procedure RandFulle(var A;n,m,numb:integer);
11:
12: implementation
13:
14: procedure LeseMass(var A;n,m:integer);
15: var
16:   i,j:integer;
17: begin
18:   for i:=1 to n do
19:     for j:=1 to m do
20:       read(GrosseMass(A)[m*(i-1)+j]);
21: end;
22:
23: procedure SchrMass(var A;n,m:integer);
24: var
25:   i,j:integer;
26: begin
```

```
27:   for i:=1 to n do
28:     begin
29:     for j:=1 to m do
30:       write(GrosseMass(A)[m*(i-1)+j],' ');
31:     writeln;
32:     end;
33: end;
34:
35: procedure RandFulle(var A;n,m,numb:integer);
36: var
37:   i,j:integer;
38: begin
39:   for i:=1 to n do
40:     begin
41:     for j:=1 to m do
42:       GrosseMass(A)[m*(i-1)+j]:=random(numb);
43:     writeln;
44:     end;
45: end;
46:
47: end.
```

Мы представили основной тип не как матрицу, а как массив, - т.е. естественный (для ПК!) способ хранения матрицы в памяти. Элемент A[i,j] матрицы размера $n \times m$ хранится в (m*(i-1)+j)-м байте от начала массива (это следует из способа хранения матрицы). Таким образом мы заполняем именно те байты, в которых хранится матрица А, которую мы передали в процедуру InputMass.

Вот так! Мы достигли желаемого результата, но при этом пожертвовали естественностью программы. В дальнейшем мы будем использовать именно модуль MatrArrU, в котором будет еще функция заполнения массивов случайными числами, но процедуры в дальнейшем будем писать с использованием более наглядной для нас матричной нотации.

Наверное, я вас уже порядочно утомил этим длиннющим разбором написания базовых функций работы с двумерными массивами. Но теперь вы уж точно хорошо разобрались в бестиповых ссылках.

10.3. Матрицы в математике

- Матрицей размера $n \times m$ называется прямоугольная таблица чисел из n строк и m столбцов. При этом элементами матрицы могут быть числа любого множества.

Обозначаются матрицы так: $A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix}$

У элементов a_{ij}: i - номер строки, а j – номер столбца, в котором находится элемент.

- Порядком квадратной матрицы называется количество ее столбцов (строк).
- Главной диагональю квадратной матрицы называется диагональ, идущая из левого верхнего в правый нижний угол.

- Побочной диагональю квадратной матрицы называется диагональ, соединяющая правый верхний и левый нижний углы.

10.4. Действия над матрицами

1. Сложение матриц

$$C = A + B = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1m} \\ b_{21} & b_{22} & \dots & b_{2m} \\ \dots & \dots & \dots & \dots \\ b_{n1} & b_{n2} & \dots & b_{nm} \end{pmatrix} = \begin{pmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \dots & a_{1m}+b_{1m} \\ a_{21}+b_{21} & a_{22}+b_{22} & \dots & a_{2m}+b_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1}+b_{n1} & a_{n2}+b_{n2} & \dots & a_{nm}+b_{nm} \end{pmatrix}$$

Т.е. для того, чтобы просуммировать 2 матрицы надо просто сложить соответствующие элементы обеих матриц, т.е. $c_{ij} = a_{ij} + b_{ij}$, где $i = \overline{1,n},\ j = \overline{1,m}$. Например:

$$\begin{pmatrix} 1 & 2 \\ 4 & 3 \end{pmatrix} + \begin{pmatrix} 3 & 5 \\ 4 & 10 \end{pmatrix} = \begin{pmatrix} 4 & 7 \\ 8 & 13 \end{pmatrix}$$

Абсолютно аналогично, если надо найти разность матриц: $C = A - B$, то элементы матрицы C будут иметь вид: $c_{ij} = a_{ij} - b_{ij}$, где $i = \overline{1,n},\ j = \overline{1,m}$.

2. Умножение матрицы на число:

$$C = sA = s \cdot \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} = \begin{pmatrix} sa_{11} & sa_{12} & \dots & sa_{1m} \\ sa_{21} & sa_{22} & \dots & sa_{2m} \\ \dots & \dots & \dots & \dots \\ sa_{n1} & sa_{n2} & \dots & sa_{nm} \end{pmatrix}$$

3. Умножение матрицы на вектор

$$C = Ax = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_m \end{pmatrix} = \begin{pmatrix} (a^{<1>}, x) \\ (a^{<2>}, x) \\ \dots \\ (a^{<n>}, x) \end{pmatrix},$$

где $(a^{<i>}, x)$ - скалярное произведение i-й строки матрицы на вектор x, т.е.:

$(a^{<i>}, x) = a_{i1}x_1 + a_{i2}x_2 + \dots + a_{im}x_m$

Например:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 1\cdot 1 + 2\cdot 2 + 3\cdot 3 \\ 4\cdot 1 + 5\cdot 2 + 6\cdot 3 \\ 7\cdot 1 + 8\cdot 2 + 9\cdot 3 \end{pmatrix}$$

4. Умножение двух матриц

$$C = A \cdot B = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1s} \\ b_{21} & b_{22} & \dots & b_{2s} \\ \dots & \dots & \dots & \dots \\ b_{m1} & b_{m2} & \dots & b_{ms} \end{pmatrix} = \begin{pmatrix} (a^{<1>}, b_{<1>}) & (a^{<1>}, b_{<2>}) & \dots & (a^{<1>}, b_{<s>}) \\ (a^{<2>}, b_{<1>}) & (a^{<2>}, b_{<2>}) & \dots & (a^{<2>}, b_{<s>}) \\ \dots & \dots & \dots & \dots \\ (a^{<n>}, b_{<1>}) & (a^{<n>}, b_{<2>}) & \dots & (a^{<n>}, b_{<s>}) \end{pmatrix},$$

где $(a^{<i>}, b_{<j>}) = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{im}b_{mj}$

Например:

$$\begin{pmatrix} 1 & 2 \\ 4 & 3 \end{pmatrix} \cdot \begin{pmatrix} 3 & 5 \\ 4 & 10 \end{pmatrix} = \begin{pmatrix} 1\cdot 3+2\cdot 4 & 1\cdot 5+2\cdot 10 \\ 4\cdot 3+3\cdot 4 & 4\cdot 5+3\cdot 10 \end{pmatrix} = \begin{pmatrix} 11 & 25 \\ 24 & 50 \end{pmatrix}$$

5. Транспонирование матрицы

- Транспонированием матрицы называется замена ее столбцов на строки, а строк на столбцы. Полученная при этом матрица называется транспонированной матрицей.

Матрица, транспонированная к матрице A, обозначается A^T.

$$A^T = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix}^T = \begin{pmatrix} a_{11} & a_{21} & \dots & a_{n1} \\ a_{12} & a_{22} & \dots & a_{n2} \\ \dots & \dots & \dots & \dots \\ a_{1m} & a_{2m} & \dots & a_{nm} \end{pmatrix}$$

Например:

1. $B = \begin{pmatrix} 1 & 3 & 4 \\ 6 & 2 & 1 \end{pmatrix} \Rightarrow B^T = \begin{pmatrix} 1 & 6 \\ 3 & 2 \\ 4 & 1 \end{pmatrix}$

Заметим, что для квадратной матрицы транспонирование эквивалентно отражению относительно главной диагонали.

Пусть вас не смущает то, что умножение матриц задается так хитро: в этом есть свой смысл. Например, следующая система линейных алгебраических уравнений

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1m}x_m = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2m}x_m = b_2 \\ \dots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nm}x_m = b_n \end{cases}$$

может быть записана очень просто:

$Ax = b$, где

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{pmatrix}, \ x = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix}, \ b = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{pmatrix}$$

Теперь, после краткого теоретического введения мы напишем программу с 2-мя процедурами: транспонирования матриц и их умножения.

Пример 4: Действия над матрицами.

```
1 : uses
2 :   MatrArrU;
3 : const
4 :   n=3; {количество строк матрицы}
5 :   m=3; {количество столбцов матрицы}
6 : type
7 : {Объявление вещественной матрицы размера n*m}
8 :   Matrix=array[1..n,1..m] of integer;
9 : var
10:   A,B,C:Matrix;
```

```
11:
12: procedure Transpon(const A:Matrix;var B:Matrix);
13: var
14:   i,j:integer;
15: begin
16:   for i:=1 to n do
17:     for j:=1 to m do
18:       B[i,j]:=A[j,i];
19: end;
20:
21: {Внимание: процедура умножения матриц будет работать
22: только, если n=m}
23: procedure MultMat(const A,B:matrix;var C:Matrix);
24: var
25:   i,j,k:word;
26: begin
27:   for i:=1 to n do
28:      for j:=1 to m do
29:        begin
30:        c[i,j]:=0; {обнуляем элемент матрицы C[i,j]}
31:        for k:=1 to n do
32:          c[i,j]:=c[i,j]+A[i,k]*b[k,j];
33:        end;
34: end;
35:
36: begin
37:   writeln('Введите матрицу A размера ',n,'*',m);
38:   LeseMass(A,n,m);
39:   writeln('Матрица A:');
40:   SchrMass(A,n,m);
41:   Transpon(A,B);
42:   writeln('Матрица B = A^t:');
43:   SchrMass(B,n,m);
44:
45:   MultMat(A,B,C);
46:   writeln('Матрица C = A*B:');
47:   SchrMass(C,n,m);
48: end.
```

10.5. Множества

Напомню, что под множеством понимается некоторая совокупность элементов, четко отличимых друг от друга.

Для множеств в TP создан специальный тип данных set. Количество элементов в типе set не может быть больше 256. Задать множество значений типа set можно так:

```
Var
  A: set of byte;
  B: set of 1..9
```

Первое определение будет означать, что значениями переменной A могут быть все подмножества множества {0, 1, …, 255}. А переменная B может быть любым подмножеством множества {1, 2, …, 9}.

Операции над множествами:

Операция	Выражение	Описание выражения
in	x in A	Принадлежит ли элемент x множеству A
<=	A<=B	Проверяет, является ли A подмножеством B
>=	A>=B	Проверяет, является ли B подмножеством A
=	A=B	Проверяет, совпадают ли A и B
<>	A<>B	Проверяет, не совпадают ли A и B
+	C:=A+B	Во множество C записывает объединение множеств A и B
-	C:=A-B	Во множество C записывает разность множеств A и B
*	C:=A*B	Во множество C записывает пересечение множеств A и B

Задавать константы множественного типа надо так:

```
const
  Alphabet=[‘a’..’z’,’A’..’Z’];
  Ziffern=[‘0’..’9’];
```

Давайте разберемся, как представляются множества в памяти ПК. Для примера рассмотрим переменную A: set of 1..8.

На нее в памяти отводится 8 бит. Каждый бит отвечает за свой элемент множества: за единицу – 1-й бит, за 2-ку второй бит, и т.д. Если элемент принадлежит множеству, то соответствующий бит равен 1, в противном случае он равен 0.

Если A=[1,2,4,8], то в памяти эти 8 бит будут выглядеть так: 11010001.

Если A=[1,4,5,8], то в памяти будет храниться 10011001.

Из такого представления сразу следует, что во множестве не может быть 2-х одинаковых элементов (тогда бит должен кодировать не только 0 или 1, но 0,1 или 2).

Хотя каждый элемент множества кодируется в памяти одним битом, но память, которая выделяется под переменную типа множество всегда должна быть кратна 8 битам (т.к. минимальной адресуемой единицей памяти является байт, а не бит). Это означает, например, что на переменную B: set of 1..9 будет выделено 2 байта.

К сожалению, в TP максимальное количество элементов множества слишком мало, – всего 256 элементов; это значительно сокращает область их применения. Однако научившись работать с динамической памятью, вы сможете на основе битового массива создать собственный тип-множество, у которого не будет столь суровых ограничений на количество элементов.

10.6. Сортировка подсчетом

Рассмотрим пример применения множеств: пусть нам надо отсортировать массив, и при этом:

1. Элементы в массиве типа byte.
2. Все элементы массива различны.
3. Количество элементов достаточно велико

Если эти 3 условия выполнены, то можно написать очень эффективный и простой алгоритм сортировки:

1. Заносим все элементы массива во вспомогательное множество.

2. Проходим по всем числам от 0 до 256 и если встречаем число во множестве, то записываем его в массив и исключаем из множества.

В результате мы расставим все числа в порядке возрастания, причем всего за два прохода, т.е. число действий должно не порядка n^2, как в сортировке пузырем или выбором, и даже не $n \log n$, как в сортировке слиянием, а только порядка n действий.

Пример 5: Сортировка подсчетом.

```
1 : const
2 :   n=5;
3 : type
4 :   Arr=array [1..n] of byte;
5 : var
6 :   M:arr;
7 :   i:byte;
8 :
9 : procedure MengeSort(var A:Array of byte);
10: var
11:   i,j:byte;
12:   S:set of byte;
13: begin
14:   S:=[];
15:   for i:=0 to High(A) do {Заполняем S элементами массива}
16:     S:=S+[A[i]];
17:   j:=0;
18:   i:=0;
19:   while (S<>[]) do {Заполняем A элементами из S}
20:     begin           {в порядке возрастания}
21:     if i in S then
22:       begin
23:       S:=S-[i];  {Убираем элемент из множества}
24:       A[j]:=i;   {и добавляем в массив}
25:       inc(j);
26:       end;
27:     inc(i);
28:     end;
29: end;
30:
31: begin
32:   for i:=1 to n do
33:     read(M[i]);
34:   writeln('Исходный массив');
35:   for i:=1 to n do
36:     write(M[i],' ');
37:   MengeSort(M);  {сортировка}
38:   writeln('Отсортированный массив');
39:   for i:=1 to n do
40:     write(M[i],' ');
41: end.
```

То, что элементы должны быть типа byte – это просто ограничение на тип set. Естественно, мы могли бы применять сортировку подсчетом, используя вместо множества массив. Если массив состоял бы из элементов типа byte, то мы могли бы использовать сортировку массивов, в которых элементы могут повторяться не более, чем 255 раз. Тогда в каждом элемента массива будет храниться то, сколько раз в исходном массиве встретилось число (отсюда и название сортировки).

Поэтому требование того, чтобы в массиве не было повторяющихся элементов – не создает больших проблем. Главное – чтобы элементы находились довольно кучно. Ведь если мы знаем, что каждое число массива находится в пределах $x \le A[i] \le y$, то нам нужен вспомогательный массив из $y - x$ элементов для того, чтобы применять эту сортировку. Если количество элементов в массиве, который мы хотим отсортировать, значительно меньше, чем $y - x$, то сортировка подсчетом будет требовать много памяти и работать очень долго. Если же кучность велика, то сортировка подсчетом будет весьма эффективна.

Самое жесткое ограничение, накладываемое на сортировку подсчетом – ограничение на тип элементов. Для целых чисел мы знаем заранее порядок следования элементов. Но если надо сортировать записи некоторого типа, порядок следования элементов которого заранее не известен, а дана лишь функция сравнения элементов, показывающая, какой из элементов должен стоять дальше, а какой ближе, то сортировка подсчетом не пройдет.

10.7. Перечисляемый тип

Описание перечисляемого типа – это список идентификаторов, заключенных в скобки, например:

```
Type
  Jahreszeiten = (Lenz, Sommer, Herbst, Winter); {Времена года}
  Farben = (Piks, Kreuze, Karos, Herzen); {Масти}
  Boolean = (False, True);
```

По большому счету, перечисленный тип – это массив констант. Для типа Jahreszeiten они равны:
Lenz=0, Sommer=1, Herbst=2, Winter=3.

Использование перечисляемого типа ограничено следующими правилами:

1. В 2-х перечисленных типах не могут использоваться одинаковые идентификаторы.

```
Type
  Typ1 = (el1,el2,el3);
  Typ2 = (el1,el2);
```

Если вы напишите эти объявления типов, то при компиляции будет выдана ошибка: {Ошибка: Duplicate identifier (el1)}

2. С переменными перечисляемого типа нельзя использовать операции Writeln, Write, readln, Read.

Использование перечисляемого типа улучшает иногда читаемость программы.

Пример 6: Тасование карт и их печать.

```
1 : uses Crt;
```

```
2 : const
3 :   n=36; {количество карт}
4 :   AFarb=4;{кол-во мастей}
5 :   AQual=9;{кол-во карт каждой масти}
6 : type
7 :   Farben=(Piks,Kreuze,Karos,Herzen);  {Масти}
8 :   Qualitat=(Sechs,Sieben,Acht,neun,zehn,Bube,Dame,Konig,As);
{достоинство}
9 :
10:   Karte=record{карта}
11:     F:Farben;
12:     Q:Qualitat;
13:   end;
14:   Kartenspiel=array[1..n] of Karte;{колода карт}
15:
16: {Заполняет массив KS картами по порядку}
17: procedure FullKartSpiel(var KS:KartenSpiel);
18: var
19:   i,j:integer;
20: begin
21:   for i:=1 to AFarb do
22:     for j:=1 to AQual do
23:       begin
24:       KS[(i-1)*AQual+j].F:=Farben(i-1);
25:       KS[(i-1)*AQual+j].Q:=Qualitat(j-1);
26:       end;
27: end;
28:
29: {Случайное заполнение колоды карт}
30: procedure ZufallKartSpiel(var KS:KartenSpiel);
31: var
32:  i,tmp:integer;
33:  Kx:Karte;
34: begin
35:   FullKartSpiel(KS);
36:   for i:=1 to n do{Меняем местами каждый элемент}
37:     begin        {с некоторым другим, выбранным случайно}
38:     tmp:=random(n)+1;
39:     Kx:=KS[i];
40:     KS[i]:=KS[tmp];
41:     KS[tmp]:=Kx;
42:     end;
43: end;
44:
45: {Печать колоды карт}
46: procedure DruckKartSpiel(var KS:KartenSpiel);
47: var
48:   i:integer;
49: begin
50:   for i:=1 to n do
51:     begin
52:     case KS[i].Q of
```

```
53:     Bube: write('B');
54:     Dame: write('D');
55:     konig: write('K');
56:     As:write('A');
57:     else {Если карта - от шестерки до десятки}
58:       write(byte(KS[i].Q)+6);
59:     end;
60:     write(chr(byte(Ks[i].F)+3),' ');{печатаем масть}
61:     end;
62: end;
63:
64: var
65:   Ksp:kartenSpiel;
66: begin
67:   Clrscr;
68:   randomize;
69:   FullKartSpiel(KSp);
70:   DruckKartSpiel(Ksp);
71:   writeln;
72:   ZufallKartSpiel(Ksp);
73:   DruckKartSpiel(Ksp);
74:   readkey;
75: end.
```

10.8. Магические квадраты

В заключение главы рассмотрим более сложный пример – построение магических квадратов.

- Магическим квадратом порядка n называется квадрат, состоящий из n^2 клеток, заполненный числами 1.. n^2 и в котором сумма элементов каждой строки, каждого столбца и обоих диагоналей равны одному и тому же числу.

Существует только 1 магический квадрат 3-го порядка (с точностью до поворотов и отражений). Его знали еще в Древнем Китае.

4	9	2
3	5	7
8	1	6

Рис 10.3. Древний китайский магический квадрат Ло Шу

Просто генерировать все возможные квадраты, а потом решать, какой из них будет магическим, а какой – нет – задача весьма нелегкая, потому что всевозможных квадратов порядка n будет $(n^2)!$ (проверьте это).

Поэтому мы будем последовательно перебирать все квадраты, при этом отбрасывая те, которые заранее не подходят.

Во-первых, можно сразу вычислить сумму элементов строки (а также столбца и диагонали) магического квадрата. Очевидно, что она равна $\frac{1+2+3+...+n^2}{n}$. В числителе – сумма элементов арифметической прогрессии, которую мы умеем находить. Т.е. магическая сумма равна $\frac{n(n^2+1)}{2}$. Эта сумма всегда будет целым числом, т.к. при любом n в числителе стоит четное число. Зная магическую сумму, мы легко сможем составить алгоритм генерирования квадратов.

Для того, чтобы мы могли на каждом шаге проверять, не нарушилось ли хотя бы одно из условий, надо хранить сумму элементов каждой строки, каждого столбца и обоих диагоналей. Кроме того, надо знать, какие числа были использованы, а какие нет. Алгоритм я написал рекурсивно.

Процедура StellNeu(i,j) пробегает все числа, которые еще не были использованы, и смотрит, какие из них можно поставить так, чтобы магические условия не нарушались. Когда функция находит такое число, она ставит его в квадрат (в матрицу A), и вызывает себя же, для того, чтобы искать подходящее число для следующего квадрата. Когда весь квадрат заполнен правильно, то процедура печатает его. Чтобы разобраться в деталях, смотрите комментарии в самом программном коде.

Пример 7: Генерация всех магических квадратов.

```
1 : uses Crt,Matrarru;
2 :
3 : const
4 :   n=3;
5 :   L=n*n;
6 : type
7 :   Matrix=array [1..n,1..n] of integer;
8 :   MasN=array[1..n] of integer;
9 :   MasL=array [1..L] of integer;
10:
11: {Печатает все магические квадраты порядка n и возвращает
12: их количество}
13: function Magisch(n:integer):longint;
14: var
15:   i,j:integer;
16:   A:Matrix;
17:   S:integer; {S - необходимая сумма элементов строки}
18:   d1,d2:integer;{Сумма элементов на диагоналях (главной и
побочной}
19:   Sp,R:MasN;{Sp[i] - сумма элем. i-го столбца, R[i] - с. эл. i-
строки}
20:   M:MasL; {M[i]=0, если число использовано}
21:   QuantMag:longint;{Храним кол-во магических квадратов}
22:
23: procedure StellNeu(i,j:integer);{Локальная процедура}
24: var
25:   k:word;
26: begin
27:   for k:=1 to l do
28:     if (M[k]<>0) and (R[i]+k<=S) and (Sp[j]+k<=S) then
```

```
29:       begin
30:       if (i=j) and (d1+k>S) then
31:         continue;
32:       if (n+1-i=j) and (d2+k>S) then
33:         continue;
34:       if (i=n) and (Sp[j]+k<>S) then
35:         continue;
36:       if (j=n) and (R[i]+k<>S) then
37:         continue;
38:
39:       if (i=n) and (j=n) then
40:         if (d1+k<>S) or (d2<>S) then
41:           continue;
42:       {Если мы дошли до этого места, значит можно ставить число}
43:       A[i,j]:=k;
44:       Sp[j]:=Sp[j]+k;
45:       R[i]:=R[i]+k;
46:       M[k]:=0;    {Число k теперь использовано}
47:       if (i=j) then
48:         d1:=d1+k;
49:       if (n+1-i=j) then
50:         d2:=d2+k;
51:       if (i=n) and (j=n) then {Заполнен весь квадрат}
52:         begin
53:         writeMass(A,n,n);
54:         inc(QuantMag);
55:         writeln;
56:         end;
57:       if j=n then      {Здесь рекурсивный вызов}
58:         StellNeu(i+1,1)
59:       else
60:         StellNeu(i,j+1);
61:       A[i,j]:=0;      {Для дальнейшего поиска обнуляем}
62:       Sp[j]:=Sp[j]-k;
63:       R[i]:=R[i]-k;
64:       M[k]:=1;
65:       if (i=j) then
66:         d1:=d1-k;
67:       if (n+1-i=j) then
68:         d2:=d2-k;
69:       end;
70: end;
71:
72: begin
73:   ClrScr;
74:   for i:=1 to n do
75:     for j:=1 to n do
76:        A[i,j]:=0;
77:   for i:=1 to L do
78:     M[i]:=1;
79:   for i:=1 to n do
80:     begin
```

```
81:      R[i]:=0;
82:      Sp[i]:=0;
83:      end;
84:    d1:=0;
85:    d2:=0;
86:    S:=n*(n*n+1) div 2;{Вычисляем магическую константу}
87:    QuantMag:=0;
88:    StellNeu(1,1);
89:    Magisch:=QuantMag;
90:  end;
91:
92:  var
93:    f:longint;
94:  begin
95:    f:=Magisch(n);
96:    writeln('Количество маг. кв. порядка ',n,' равно ',f);
97:  end.
```

Если честно, то квадраты 4-го порядка эта программа еще выдает за обозримое время, но с магическими квадратами 5-го порядка дело идет совсем туго. Можете попытаться модернизировать алгоритм, сделав его итерационным и таким, который бы учитывал симметрию. Однако программа все равно будет работать долго – задача слишком сложна.

Задачи

1. Пусть x – n-мерный массив, A,B,C – квадратные матрицы порядка n. Вычислить: а) C=A+B; б) y=Ax;
2. Найти сумму элементов матрицы.
3. Найти максимальный и минимальный элемент в матрице.
4. Переставить первый и последний столбцы матрицы.
5. Дана прямоугольная матрица A размера $m \times n$ и массив B логических переменных размера m. По матрице A получить массив B, присвоив его k-му элементу значение true, если элементы k-й строки матрицы A упорядочены по убыванию, и false в противном случае.
6. Дана прямоугольная вещественная матрица. Упорядочить ее строки по неубыванию:
   а) первых их элементов;
   б) суммы их элементов;
   в) их наибольших элементов.
7. Определить, является ли заданная матрица симметрической (матрица A симметрическая, если $A = A^T$).
8. Элемент матрицы назовем его седловой точкой, если он является наименьшим в своей строке и одновременно наибольшим в своем столбце или, наоборот, является наибольшим в своей строке, и наименьшим в своем столбце. Для заданной целой матрицы напечатать индексы всех ее седловых точек (печатать в виде (i,j)).
9. Дана квадратная вещественная матрица, все элементы которой различны. Найти скалярное произведение строки, в которой находится наибольший элемент матрицы на столбец с наименьшим элементом.
10. Пусть вам дана таблица шашечного турнира:

№	Имя Участника	Номер соперника			Общий счет
		1	2	3	
1	Иванов		2	1	3
2	Петров	0		1	1
3	Сидоров	1	1		2

Судье захотелось поменять порядок следования участников турнира на противоположный (как показано в таблице 2).

№	Имя Участника	Номер соперника			Общий счет
		1	2	3	
1	Сидоров		1	1	2
2	Петров	1		0	1
3	Иванов	1	2		3

Напишите программу, которая по данной таблице встреч участников печатает эту же таблицу, но для обратного порядка следования участников.

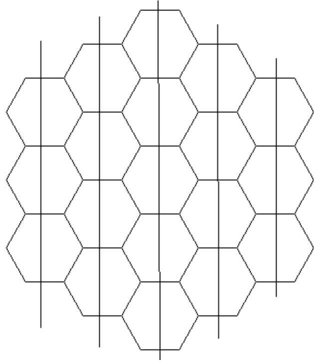
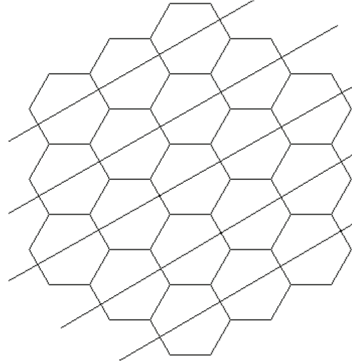
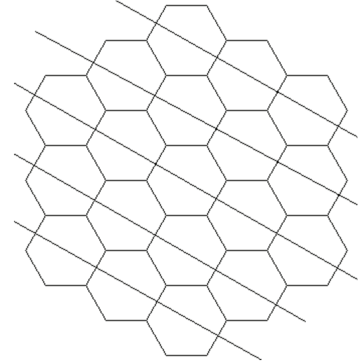
(в данном случае входная матрица $\begin{pmatrix} & 2 & 1 \\ 0 & & 1 \\ 1 & 1 & \end{pmatrix}$, выходная - $\begin{pmatrix} & 1 & 1 \\ 1 & & 0 \\ 1 & 2 & \end{pmatrix}$. (диагональные элементы можно заполнить нулями).

11. Обобщите предыдущую задачу на случай того, если новый порядок следования игроков может быть произвольный, и задается некоторой подстановкой (т.е. строки (1,2,…,n) переходят в строки $(i_1, i_2, ..., i_n)$, например, если $(1, 2, ..., n) \to (n, n-1, ..., 1)$, то получим предыдущую задачу).
12. Дана квадратная матрица порядка n. Заполните по спирали матрицу числами от 1 до n^2. Спираль должна начинаться в верхнем левом углу матрицы, и заканчиваться в ее центре. Вид решения для n=5 таков:

$$\begin{matrix} 1 & 2 & 3 & 4 & 5 \\ 16 & 17 & 18 & 19 & 6 \\ 15 & 24 & 25 & 20 & 7 \\ 14 & 23 & 22 & 21 & 8 \\ 13 & 12 & 11 & 10 & 9 \end{matrix}$$

13. Бинарный поиск в матрице. Дана матрица размера $n \times m$, упорядоченная по строкам и столбцам: $x[i, j] \le x[i+1, j]$, $x[i, j] \le x[i, j+1]$. Требуется найти в матрице элемент a. Количество действий должно быть порядка $n + m$.
14. Дан массив с элементами типа byte. Надо узнать количество различных элементов в нем. Количество действий должно быть порядка cn, где n – длина массива.
15. Вычислить все простые числа в диапазоне 0..255 с помощью решета Эратосфена. Алгоритм заключается в следующем: Берем первое число – двойку, и вычеркиваем все числа, кратные двум, кроме самой двойки, затем берем наименьшее из невычеркнутых чисел, больших двух, и вычеркиваем все кратные ему, и т.д., пока из множества не будут вычеркнуты все ненужные элементы.
16. Обобщить метод решета Эратосфена так, чтобы можно было вычислять простые числа в большем диапазоне.

17. Пусть M - магический квадрат порядка n, составленный из чисел $1,2,...,n^2$. Обозначим через M_k магический квадрат, получающийся заменой каждого числа m на $m+(k-1)n^2$. Докажите, что заменив каждое число k на магический квадрат M_k, получим магический квадрат порядка n^2.
18. В 1910 году Клиффорд У. Адамс принялся на поиски магического шестиугольника 3-го порядка. Адамс взял 19 керамических плиток с числами от 1 до 19 и начал составлять из них различные шестиугольники, пытаясь найти магический, т.е. такой, чтобы сумма чисел в каждом ряду из прилегающих друг к другу шестиугольников была одинакова (все ряды приведены на следующих 3-х рисунках). 47 лет Адамс пытался найти такой шестиугольник, пока наконец в 1957 году не нашел решения. Он начертил его на бумаге, но потом потерял ее и еще 5 лет восстанавливал решение. Удивительно, но впоследствии выяснилось, что этот шестиугольник единственный (с точностью до поворотов и отражений) шестиугольник 3-го порядка, а также то, что шестиугольников порядка не равного 3 не существует вообще (кроме, естественно, шестиугольника 1-го порядка). Вам предлагается сделать то, над чем бедный Адамс трудился 52 года. Посмотрим, за сколько времени с этой задачей справитесь вы.

Проект 2: Теория голосования

Пусть идут выборы президента, и на высшую государственную должность есть 4 претендента: a, b, c, d. Был проведен социологический опрос, который показал, что всех избирателей (80 человек, или, если хотите, миллионов человек) можно разбить на 5 групп:

I	II	III	IV	V
15	11	23	21	10
a	*b*	*c*	*a*	*d*
b	*d*	*d*	*d*	*b*
d	*c*	*b*	*b*	*c*
c	*a*	*a*	*c*	*a*

Первый столбик означает: 15 избирателей считают, что a – самый лучший кандидат, за ним идет b, далее – d, а кандидат c – самый плохой.
Наша цель узнать, кто из кандидатов станет победителем, если результаты соцопроса подтвердятся при голосовании.

Давайте считать, что победителем на выборах будет тот кандидат, который набрал наибольшее количество голосов в 1-м туре (такая процедура называется **правилом относительного большинства**). Для данной таблицы победителем будет кандидат A, т.к. за него проголосуют 36 кандидатов, а за ближайшего из претендентов, C, проголосуют лишь 23 избирателя.

Это – простейшая система выборов, которая, тем не менее, применяется очень часто.

Давайте рассмотрим вторую распространенную систему избрания президента, которая используется, например, в Украине. Эта система называется **правило голосования с последовательным исключением**. Согласно этой системе если в первом туре ни один из кандидатов не набрал абсолютного большинства голосов (свыше 50%), то проводится второй тур, в котором участвуют 2 кандидата, набравшие в первом туре наибольшее количество голосов, и победитель второго тура и будет победителем выборов.

Давайте проанализируем результаты соцопроса. Во второй тур выйдут кандидаты A и C, а голоса избирателей, считавших, что наилучшие из кандидатов – B и D разделятся между ними. Так как и вторая, и четвертая группы избирателей считают, что кандидат C лучше, чем A, то они и проголосуют за C. В результате во втором туре кандидат A наберет 36 голосов, а кандидат C – 44 голоса. Победителем будет C!

Выходит, что при одних и тех же настроениях избирателей победителями будут разные кандидаты. Сталин говорил: «Главное не то, как проголосовали, а то, как подсчитали». И это - правда! Более того, большинство избирателей считают A самым худшим из кандидатов, и C тоже не сильно любят, а ведь именно они и стали победителями. Очевидно, что надо придумать методику подсчета голосов, учитывающую степени симпатии/антипатии избирателей к кандидатам.

Первая попытка критического анализа процедур голосования была предпринята лишь в конце XVIII века во Франции. Основоположниками теории голосования являются 2 ученых, членов Парижской АН: Мари Жан Антуан Никола Корита, маркиз де Кондорсе и Жан Шарль де Борда.

Давайте рассмотрим правило, предложенное Борда для таблицы голосования, используемой ранее.

	I	II	III	IV	V
	15	11	23	21	10
3	*a*	*b*	*c*	*a*	*d*
2	*b*	*d*	*d*	*d*	*b*
1	*d*	*c*	*b*	*b*	*c*
0	*c*	*a*	*a*	*c*	*a*

Согласно этой системе за последнее место дается 0 баллов, за предпоследнее – 1 балл, и так далее. Затем считаем сумму очков для каждого из кандидатов. Тот, который набрал наибольшее количество баллов, и объявляется победителем. Давайте подсчитаем:

$B_A = 15 \cdot 3 + 0 \cdot 11 + 0 \cdot 23 + 21 \cdot 3 + 0 \cdot 10 = 108$

$B_B = 30 + 33 + 23 + 21 + 20 = 127$

$B_C = 11 + 69 + 10 = 90$

$B_D = 185$.

Итак, согласно процедуре Борда, победитель – кандидат D, причем количество баллов, набранных им, более чем в 2 раза превосходит количество баллов, полученных C.

Теперь у нас уже три разных победителя при тех же настроениях зрителей, так что справедливость той или иной процедуры подсчета голосов – понятие весьма относительное.

Сейчас я приведу в качестве примера модуль с некоторыми вспомогательными процедурами, а затем мы рассмотрим еще несколько систем обработки голосов избирателей.

Пример: Процедура случайного заполнения матрицы голосования.

```
unit HelfAbst;

interface
const
  n=3;{Количество кандидатов}
  m=6;{Количество групп избирателей}
type
  Matrix=array[1..n,1..m] of integer;{матрица голосования}
  Gruppen=array[1..m] of integer;
  Kandidaten=array[1..n] of integer;

{gleich - одинаковый, die Spalte - столбец
(сравните с украинским шпальта (колонка)
Возвращает true, если в матрице A столбцы ind1, ind2 одинаковы}
function SindGleichen(var A:Matrix;ind1,ind2:integer):boolean;
{Заполняет матрицу голосования случайными группами}
procedure RandMat(var A:Matrix);
{Находит максимальное число в массиве (первое попавшееся)}
function max(var K:array of integer):integer;

implementation
```

```
function SindGleichen(var A:Matrix;ind1,ind2:integer):boolean;
var i:integer;
begin
  for i:=1 to n do
    if (A[i,ind1]<>A[i,ind2]) then
      begin
      SindGleichen:=false;
      exit;
      end;
  SindGleichen:=true;
end;

procedure RandMat(var A:Matrix);
var
  i,j,k:integer;
  flag:boolean;
  ind,ind2,tmp:integer;
begin
  for j:=1 to m do
    begin
      for i:=1 to n do{Заполняем по порядку}
        A[i,j]:=i;
      for i:=1 to n do {Случайно переставляем все элементы}
        begin
        ind:=random(n)+1;{Получаем место, куда переставить текущий
элемент}
        tmp:=A[i,j];    {Переставляем }
        A[i,j]:=A[ind,j]; {i-й и ind-й элементы j-го столбца}
        A[ind,j]:=tmp;
        end;
      repeat {Проверяем на совпадение с некоторым из столбцов}
        flag:=true;{Считаем, что совпадений нет}
        for k:=1 to j-1 do
          if SindGleichen(A,k,j) then
            begin
            flag:=false; {Нашли одинаковые столбцы}
            ind:=random(n)+1;
            ind2:=random(n)+1;
            tmp:=A[ind,j];   {Меняем местами 2 случайных элемента}
            A[ind,j]:=A[ind2,j];
            A[ind2,j]:=tmp;
            break;
            end;
       until flag=true;
    end;
end;

function max(var K:array of integer):integer;
var
  i,ind,tmp:integer;
begin
```

```
  tmp:=K[0];
  ind:=0;
  for i:=0 to High(K) do
    if K[i]>tmp then
      begin
      tmp:=K[i];
      ind:=i;
      end;
  max:=ind+1;{+1 т.к. индексация внутри процедуры смещена на -1}
end;
end.
--------------------основной файл----------------------
uses Crt,
     MatrArru,{для процедур InputMass и WriteMass}
     HelfAbst;
var
  Mat:Matrix;
  G:Gruppen;

{Процедура относительного большинства}
function Beste1Tur(var A:Matrix;var G:Gruppen):integer;
var
  K:Kandidaten;
  i,j:integer;
begin
  for i:=1 to n do
    K[i]:=0;
  for j:=1 to m do
    K[A[1,j]]:=K[A[1,j]]+G[j];
  Beste1Tur:=Max(K);
end;
begin
  clrscr;
  randomize;
  writeln('enter A');
  RandMat(Mat);
  WriteMass(Mat,n,m);
  writeln('enter G');
  InputMass(G,1,m);
  writeln('Победитель: ',Beste1Tur(Mat,G));
  readkey;
end.
```

Думаю, что вы, как опытные программисты, уже можете легко разобраться в этом модуле самостоятельно. Однако необходимо отметить, что функция Max (использующаяся в Beste1Tur) возвращает первое из максимальных значений, встретившихся в массиве, что не соответствует постановке задачи: нам надо, чтобы процедура находила наибольшее значение (а оно может быть лишь одно), либо уведомляла о том, что наибольшего элемента не существует.

Теперь давайте рассмотрим еще 2 правила определения победителя выборов.

Обобщенное правило Борда. В правиле Борда худшему кандидату давалось 0 очков, и каждому следующему – на один балл больше. Но при большом числе кандидатов

избиратели просто не знают многих из них, и расставят всех, за исключением нескольких наилучших и наихудших, наугад. Обобщенное правило Борда учитывает этот недостаток: за i-е место в дается s_i баллов. Причем $s_1 \geq s_2 \geq ... \geq s_n$, и $s_1 > s_n$, где n - количество кандидатов.

Процедура Кондорсе: победителем по Кондорсе называется кандидат, который побеждает любого другого кандидата при парном сравнении по правилу большинства.

Часто бывает, что победителя по Кондорсе вообще нет. Такая ситуация называется парадоксом Кондорсе.

Задачи по теории голосования

1. Изменить функцию Max, чтобы процедура определения победителя для правила относительного большинства работала правильно.
2. Написать процедуру определения победителя для голосования с последовательным исключением.
3. Написать процедуру определения победителя для обобщенного правила Борда.
4. Написать процедуру определения победителя для процедуры Кондорсе.
5. Придумайте свою систему выявления победителя и напишите соответствующую процедуру к ней.
6. Представьте, что президент некоторой страны - хитрый и нечистый на руку человек. По его заказу был проведен широкомасштабный соцопрос. У этого президента есть возможность заставить председателя Центральной Избирательной Комиссии выбрать ту процедуру голосования, какую он посчитает нужной. Вы – главный программист того кандидата, который является наиболее достойным из тех, кто собирается занять президентское кресло и поэтому вы должны написать программу, которая определит, какую из процедур голосования (всех приведенных мною, исключая обобщенное правило Борда) выберет этот президент-хитрюга, чтобы ваш кандидат знал, где ждать подвоха, и какие аргументы нести в Конституционный суд.
7. Представьте, что этот президент из предыдущей задачи так надоел народу, что ни одна из процедур не спасает его от поражения. Для того, чтобы не прибегать к фальсификации выборов и насилию, он хочет испытать последний шанс – попытаться подобрать коэффициенты к обобщенной процедуре Борда так, чтобы он все-таки был избран на второй срок. Вы должны проверить и эту возможность, и попытаться найти набор коэффициентов (хотя бы один), который обеспечит победу нынешнему президенту.

Глава 11: Символы и строки

11.1. Массивы символов

С символами вы уже умеете работать. Разумеется, можно использовать и массивы с элементами типа char. По большому счету, строка – это и есть просто набор следующих друг за другом символов, поэтому массив с элементами типа char можно интерпретировать и как строку. Однако это очень неудобно, и сейчас вы увидите, почему.

Пример 1: Массив char'ов как строки.

```
1 : uses Crt;
2 : const n=10;
3 : type
4 :   Chmas=array[1..n] of char;
5 : var
6 :   A:Chmas;
7 :   i:byte;
8 : begin
9 :   clrscr;
10:   writeln('Введите строку:');
11:   for i:=1 to n do
12:     read(A[i]);
13:   writeln('Результат');
14:   for i:=1 to n do
15:     write(A[i]);
16: end.
```

Ничего странного в этой программе нет. Все проблемы возникают при вводе элементов массива: если ваша строка = 5 символам, вы все равно должны вводить все n, если вы заканчиваете ввод строки клавишей Enter, то он тоже считается символом, что зачастую неудобно.

11.2. Тип String

Тип string позволяет обрабатывать строки гораздо эффективнее, чем массив char'ов. Преимущества string: с переменными этого типа можно работать и как с обычными массивами, и при этом использовать встроенные функции для работы со строками а также операции конкатенации, сравнения и т.д. Встроенные функции для обработки строк работают быстро, т.к. они написаны на ассемблере, в котором предусмотрены специальные цепочечные операции, с помощью которых работа со строками проводится очень быстро.

Фактически, переменная типа string – последовательность символов, длина которой не превышает 256 символов. При этом в 0-м байте хранится символ, ASCII-код которого равен количеству символов в строке.

Объявления переменных:

```
Var
  St1:string;
  St2:string[20];
```

Под переменную st1 резервируются 256 байт, а под переменную st2 – только 21.

К строкам можно применять операции +, <, >, <=, >=, =, <>.

\+ - операция конкатенации (сцепления).

Str1:=str1+str2; означает, что к строке str1 присоединяется строка str2.

Если в переменную s типа string присвоить строку, длина которой превышает длину переменной s, то все лишние символы будут отброшены.

Сравнения строк производятся поэлементно, слева направо, с учетом кодировки ASCII.

Например:

‘Apple’>’Apfel’ (код ‘f’ меньше, чем код ‘p’)

‘111’<’a11’ (код ‘1’ меньше, чем код ‘a’)

Основные подпрограммы для работы со строками:

- **function Copy (st:string; index,numb:integer):string;**
  Копирует из строки st numb символов, начиная с номера index.
- **procedure Delete(var st:string;index,numb:integer);**
  Удаляет из строки st numb символов, начиная с номера index.
- **function Length(st:string):integer;**
  Возвращает длину строки, т.е. количество символов в ней (а не объем памяти, которая была выделена под переменную).
- **procedure Insert(sub:string; var st:string;index:integer);**
  Вставляет строку sub в строку st, начиная с номера index.
- **function Pos(SubS,S:string):byte;**
  Возвращает позицию, начиная с которой в строке S располагается подстрока SubS (0 – S не содержит SubS).

Пример 2: Простейшие приемы работы со строками.

```
1 : unit ZeilenU; {Zeile = строка}
2 :
3 : interface
4 :
5 : function IstKorrekt (const st:string):boolean;
6 : {Функция, проверяющая, правильно ли стоят в строке скобки}
7 : function ZeilUpCase (const st:string):string;
8 : {Функция, которая возвращает строку, у которой все строчные
9 : латинские буквы заменены на заглавные}
10: function Umdrehung(const st:string):string;  {Umdrehung =
переворот}
11: {Функция, которая возвращает "перевернутую" строку}
12:
13: implementation
14:
15: function Umdrehung(const st:string):string;
16: var
17:   h,i:byte;
18: begin
19:   h:=length(st);
```

```
20:   for i:=1 to h do
21:     Umdrehung[i]:=st[h+1-i];
22:   Umdrehung[0]:=chr(h);
23: end;
24:
25: function ZeilUpCase (const st:string):string;
26: var
27:   i:byte;
28: begin
29:   for i:=1 to length(st) do
30:     ZeilUpCase[i]:=UpCase(st[i]);
31: end;
32:
33: function IstKorrekt (const st:string):boolean;
34: var
35:   h,i:byte;
36:   n:integer; {n - разность между количеством открывающих}
37:       {и закрывающих скобок}
38: begin
39:   h:=length(st);
40:   n:=0;
41:   for i:=1 to h do
42:     begin
43:     if st[i]=')' then  {разность стала на один меньше}
44:       dec(n);
45:     if st[i]='(' then  {разность стала на один больше}
46:       inc(n);
47:     if n<0 then   {Если кол-во закр. скобок стало больше числа}
48:       begin       {открывающих, то скобки расставлены неверно}
49:       IstKorrekt:=false;
50:       exit;
51:       end;
52:     end;
53:   if n=0 then   {Если кол-во откр. скобок = числу закр., то }
54:     IstKorrekt:=true {строка - верна}
55:   else
56:     IstKorrekt:=false;
57: end;
58: end.
```

Далее – основная программа

```
1 : uses ZeilenU;
2 : var
3 :   str1,str2:string;
4 : begin
5 :   read(str1);
6 :   str2:=Umdrehung(str1);{Переворачиваем строку str1}
7 :   writeln;
8 :   writeln(str2);
9 :   str2:=ZeilUpCase(str2);{Переводим str2 в верхний регистр}
10:   writeln(str2);
11:
```

```
12:   str1:='Mathematiker';
13:   str2:=' und Programmierer';
14:   str1:=str1+str2; {Mathematiker und Programmierer}
15:   writeln(str1);
16:
17:   str2:='Sie sind ';
18:   Insert(str2,str1,1); {Sie sind Mathematiker und Programmierer}
19:   writeln(str1);
20:   Delete(str1,5,10);
21:   writeln(str1);
22:
23:   str1:='((ddd)dddd(';
24:   writeln(str1,' ',IstKorrekt(str1)); {false}
25:   str1:='(ddd) sss (hh)';
26:   writeln(str1,' ',IstKorrekt(str1)); {true}
27:   readln;
28: end.
```

Принцип проверки строки на правильность расстановки одного типа скобок прост: в специальной переменной хранится разность между количеством открытых и закрытых скобок. Если на каком-то символе это число станет <0, то скобки расставлены неверно. В противном случае скобки расставлены правильно.

Мы написали процедуру, проверяющую правильность расстановки скобок. Но скобки могут быть разных типов: (), {}, [], “”, ‘’ и т.д. Нашей целью теперь станет проверить правильность расстановки нескольких типов скобок. Просто проверить на правильность расстановки каждую из них по очереди нельзя: это еще не гарантирует правильность написания строки (пример такой строки: « (aaa{ffff)} »). Следовательно, воспользоваться предыдущей процедурой нельзя – хотя она и выглядит красиво и миниатюрно, но она не может стать основой для более общего алгоритма.

Решать задачу будем так: проходим по строке и заносим индексы открывающих скобок в массив. Если же встречается закрывающая скобка, то если она соответствует ближайшей открывающей скобке, то удаляем открывающую скобку из массива, в противном случае скобки в строке расставлены неверно. Если достигнут конец строки, то, если количество открывающих скобок в массиве равно нулю, значит скобки в строке расставлены правильно. В противном случае строка записана неверно.

Пример 3: Проверка правильности расстановки нескольких типов скобок.

```
1 : const
2 :   n=3;
3 : type
4 :   Klammern = array [1..n,1..2] of char;  {Klammer = скобка}
5 :
6 : { k=1 =>Проверяет, явл. ли символ с открывающей скобкой }
7 : { k=2 =>Проверяет, явл. ли символ с закрывающей скобкой }
8 : function IstKlammer(c:char;k:byte;var D:Klammern):byte;
9 : var
10:   j:byte;
11: begin
12:   for j:=1 to n do
13:     if (c=D[j,k]) then
```

```
14:       begin
15:       IstKlammer:=j;
16:       exit;
17:       end;
18:   IstKlammer:=0;
19: end;
20:
21: function IstKorrekt(const s:string;var Divis:Klammern):boolean;
22: var
23:   Stapel:array [1..255] of byte;  {Stapel = стек}
24:   KlQuant:byte; {Klammern Quantitat = количество скобок}
25:   i,indB:byte;
26: begin
27:   KlQuant:=0;
28:   i:=1;
29:   while i<=length(s) do
30:     begin
31:     indB:=IstKlammer(s[i],1,Divis);{Является ли открывающей ск.}
32:     if indB<>0 then
33:       begin
34:       inc(KlQuant);  {Увеличиваем количество открытых скобок}
35:       Stapel[KlQuant]:=i;{Заносим индекс скобки в стек}
36:       inc(i);
37:       continue;
38:       end;
39:     indB:=IstKlammer(s[i],2,Divis);{Является ли закрывающей ск.}
40:     if indB<>0 then {Нашли закрывающую скобку}
41:       begin
42:       if KlQuant=0 then{Если открывающих нет}
43:         begin
44:         IstKorrekt:=false;
45:         exit;
46:         end;
47:       if (Divis[indB,1]=s[Stapel[KlQuant]]) then{Если скобка
подходит}
48:         dec(KlQuant){уменьшаем кол-во открытых скобок в стеке}
49:       else
50:         begin
51:         IstKorrekt:=false;
52:         exit;
53:         end;
54:       end;
55:     inc(i);
56:     end;
57:
58:   if KlQuant=0 then {Если скобок нет}
59:     begin
60:     IstKorrekt:=true;
61:     exit;
62:     end;
63:   IstKorrekt:=false;
64: end;
```

```
65:
66: var
67:   s:string;
68: const
69:   D:Klammern=(('{','}'),('(',')'),('[',']'));{типизир.
константа}
70: begin
71:   writeln('Введите строку:');
72:   readln(s);
73:   writeln(IstKorrekt(s,D));
74:   readln;
75: end.
```

В строке 69 объявлена типизированная константа (ТК) типа Divisors. ТК – это на самом деле переменная, которая инициализирована начальным значением. Чтобы объявить ТК простого типа (числа, символы, строки, булевский тип), например, целого, надо записать так:

```
const  a:integer = 5;
```

Чтобы сделать массив А из четырех вещественных чисел типизированной константой, надо перечислить элементы, заключив их в скобки:

```
A:array[1..4] of real = (1.2, 2.3, 3.4, 4.1)
```

Если массив двумерный (как в примере), то каждая из компонент двумерного массива представляет собой массив, и должна быть соответствующим образом записана.

Если типом является запись, то надо в скобках перечислить значения каждого из полей.

Как внести в строку символы, которые нельзя написать на клавиатуре

Для того чтобы вставить в строку спец. символы, вы можете использовать функцию chr, которая по коду вернет вам любой символ из набора символов ASCII, или использовать машинные коды. Чтобы получить символ с нужным машинным кодом, надо набрать символ #, а затем – число.

Пример 4: Использование машинных кодов для написания спец. символов.

```
begin
  write('Привет! '+#13#10+'А это - другая строка');
  write(chr(65)+chr(66));
  readln;
end.
```

Результат работы этой программы такой:

Привет!

А это – другая строка

AB

Задачи

1. Дан массив символов. Надо все символы ‘w’, встречающиеся в строке заменить пробелами.
2. Сосчитать количество слов в массиве символов (слово – последовательность символов, перед которой и после которой стоит пробел).
3. Дан массив символов из 26 элементов. Случайным образом заполнить все его элементы строчными буквами латинского алфавита.
4. Дан массив символов из 26 элементов. Случайным образом заполнить все его элементы строчными буквами латинского алфавита так, чтобы ни одна из букв не повторялась дважды.
5. Дан массив символов. Проверить, можно ли из букв этого массива построить заданное слово. Буквы из массива могут использоваться лишь по одному разу. Например, из слова “Математик” слово “мама” составить можно, а слово «папа» нельзя.
6. Дана строка. Сосчитать количество пробелов в ней.
7. Дана строка. Если в строке будет идти подряд несколько одинаковых символов, то надо удалить все, кроме одного.
8. Дана строка. Найти в ней все слова, начинающиеся с буквы a, и вывести их на экран.
9. Дана строка. Найти количество различных символов в ней.
10. Дан массив строк. Каждая строка – английское слово. Требуется отсортировать этот массив в алфавитном порядке.
11. Проверить, является ли введенная фраза палиндромом, без учета пробелов (например, «Аргентина манит негра»).
12. Вам дана строка, в которой написано целое число. Вы должны записать в переменную типа longint число, записанное в строке.
13. Напишите программу, единственным действием которой было бы очистить экран и напечатать на дисплей саму себя (т.е. программный код). Использовать файл с исходным кодом нельзя.

Глава 12: Файлы

Во всех программах, которые мы писали до сих пор, нельзя было сохранять результаты их работы, а также загружать дополнительную информацию из файлов. В этой главе мы научимся это делать.

12.1. Логические и физические файлы.

В TP есть 2 типа файлов: логические и физические.

- Физический файл – именованная область энергонезависимой памяти.
- Каталог – системный файл, поддерживающий структуру ФС.

К физическим файлам, кроме «обыкновенных» файлов, в которых хранятся данные, относятся некоторые устройства, например: принтер, клавиатура.

- Логические файлы – это переменные одного из файловых типов TP.

Всего таких типов 3:

- Текстовые файлы (типа Text);
- Типизированные файлы (типа File of <некоторый тип данных>)
- Бестиповые файлы (типа File)

Текстовые файлы состоят из кодов ASCII, включая управляющие и расширенные коды.
Знак конца строки – символ с кодом 13 (#13).
Знак перевода строки – символ с кодом 10 (#10).
Знак конца файла – символ с кодом 26 (#26).

Вся информация в текстовом файле представлена в виде символов, которые ее изображают.

Типизированные файлы состоят из машинных представлений того типа данных, который стоял в заголовке описания файловой переменной. Бестиповые файлы тоже состоят из машинных представлений данных, но их тип не определен заранее и в них можно комбинировать данные разных типов.

12.2. Открытие и закрытие файлов.

Прежде, чем начать работу с физическим файлом, надо связать его с каким-то логическим файлом.

Это можно сделать с помощью процедуры assign(var f; path:string);
f – переменная любого файлового типа,
path – путь к физическому файлу.

Например:

```
assign(f,'C:\Programmierung\smth.pas');
```

Если файл находится в текущем каталоге, то весь пусть к файлу можно не указывать, например:

```
assign(f,'aaa.txt');
```

После того, как физический файл связан с логическим, этот файл можно открыть файл для записи или чтения данных.

- Rewrite(var f), где f – некоторый логический файл, создает файл f и открывает его для записи данных.
- Reset(var f) открывает файл для чтения данных (но не создает его!).
- Close(var f) закрывает открытый ранее файл. Если процедуру close вызвать для уже закрытого файла, то произойдет ошибка в программе.

12.3. Текстовые файлы

Пример 1: Программа, позволяющая записать несколько строк в файл.

```
1 : var
2 :   f:Text;
3 :   s:string;
4 : begin
5 :   assign(f,'file1.txt');{связываем файл f с физическим файлом}
6 :   rewrite(f);{открываем f}
7 :   repeat
8 :     readln(s);
9 :     if s='' then {Пустые строки не печатаем}
10:       break;
11:     writeln(f,s);{записываем в файл f строку s}
12:   until s='';{пока не будет введена пустая строка}
13:   close(f);{если файл не закрыть, то данные потеряются}
14: end.
```

В этой программе вы будете вводить строки в файл, а они будут выводиться в файл 'file1.txt'. Ввод прекратится, как только вы введете пустую строку.

Запись строк в файл производится с помощью знакомых нам процедур write и writeln. Компилятор анализирует первый аргумент функции, и если это типизированный или текстовый файл, то он понимает, что все последующие аргументы надо записывать в файл f. Абсолютно аналогичные свойства у процедур read и readln.

Пример 2: Создание файлов, в которых хранятся массивы чисел.

```
1 : const
2 :   n=15;
3 : var
4 :   f,f2:Text;
5 :   M:array[1..n] of real;
6 :   i:integer;
7 :   x:real;
8 : begin
9 :   assign(f,'file2.txt');{связываем файл f с физическим файлом}
10:   rewrite(f);{открываем f}
11:   for i:=1 to n do  {заполняем файл случайными числами}
12:     writeln(f,random*100);
13:   assign(f2,'file22.txt');
14:   rewrite(f2);{открываем f2}
15:   reset(f);{открываем f для чтения}
16:   for i:=1 to n do {заполняем массив числами из файла}
17:     readln(f,M[i]);
18:   for i:=n downto 1 do{заполняем числами файл f2}
```

```
19:     writeln(f2,M[i]);
20:   close(f);{А сейчас закрыть оба файла надо обязательно}
21:   close(f2);
22: end.
```

Как видите, в строке 10 файл f открывается для записи, а в строке 15 открывается для чтения. Когда вы работаете с файлами, вы можете сколько угодно раз открывать их для чтения или записи, не закрывая их при этом. Однако в конце работы программы все открытые файлы должны быть обязательно закрыты.

Для того, чтобы добавлять информацию в конец файла (существующего), не стирая содержащейся в ней до этого информации, есть специальная процедура:

```
Append(var f:Text)
```

Пример 3: Добавление строк в текстовый файл.

```
1 : var
2 :   f:Text;
3 :   s:string;
4 : begin
5 :   assign(f,'file1.txt');{связываем файл f с физическим файлом}
6 :   append(f);{открываем f для дозаписи}
7 :   repeat
8 :     readln(s);
9 :     writeln(f,s);{записываем в файл f строку s}
10:   until s='';{пока не будет введена пустая строка}
11:   close(f);{если файл не закрыть, то данные потеряются}
12: end.
```

Эта программа ничем не отличается от примера №1, кроме строки №6 (вместо процедуры Rewrite используется Append).

В примере №2 мы читали из файла массив чисел. Но при этом мы считывали заранее известное количество чисел. Но ведь обычно заранее не известно, сколько чисел или символов в файле; в таких случаях надо знать, когда заканчивать чтение информации из файла. Для такой цели есть специальная функция EOF(var f) (EOF – сокращение End Of File), которая возвращает true, если достигнут конец файла, а false – в противном случае.

Для текстовых файлов есть еще одна полезная функция – Eoln(var f:Text), возвращающая true, если достигнут конец строки, false – в противном случае.

12.4. Типизированные файлы

Если вы хоть раз посмотрели в файлы, с которыми мы работали в примерах 1..3, то вы поняли, что писать в текстовый файл – то же самое, что печатать просто на экран: если вы записываете в файл переменную x, в которой хранится число 112, то и в файле запишется «112», если пишете переменную a, в которой хранится строка ‘Anders’, то и в файле будет написано Anders.

При работе с типизированными файлами принцип другой: если вы печатаете значение переменной, то в файл будет занесена та информация, которая хранится в памяти под именем вашей переменной. Давайте рассмотрим простой пример, чтобы вы все хорошо поняли.

Пример 4: Типизированный файл из целых чисел.

```
1 : var
2 :   f:file of integer;
3 :   i:integer;
4 : begin
5 :   assign(f,'fileInt.txt');
6 :   rewrite(f);
7 :   for i:=24897 to 24897+4 do
8 :     write(f,i);
9 :   close(f);
10: end.
```

Итак, мы хотим создать file of integer, и записать туда 5 чисел: 24897, 24898, 24899, 24900, 24901.

Переменная integer содержит 2 байта, поэтому на любое число типа integer в file of integer будет отводиться 2 байта. Давайте подсчитаем, что нам запишет компьютер в файл. Двоичное представление числа 24897 такое: 01100001 01000001. Т.е. в старшем байте находится число 01100001 (в десятичной ССч 97), а в младшем – 01000001 (в десятичной ССч 65). При записи в типизированный файл ПК каждый байт интерпретирует как символ, и записывает ASCII-код, соответствующий числу, которое в байте хранится. Числу 1 соответствует символ «а», а числу 65 – символ «А».

Кроме того, в переменной, которая содержит 2 байта, младший и старший байты хранятся в обратном порядке (вы уже сталкивались с этим в главе «Матрицы»). В итоге получим, что представление числа 24897 будет таким: Aa.

А все 5 чисел запишутся в файле так: AaBaCaDaEa.

Если вам надо прочитать из file of integer несколько чисел, то компьютер будет извлекать по 2 символа, и переводить их в соответствующее число.

Какие можно сделать выводы:

- В типизированных файлах очень экономно хранятся числа.
- Читать типизированные файлы можно, лишь если вы знаете, какого типа информация в нем хранится (т.е. знаете формат файла), и как хранится эта информация в ПК.

12.5. Последовательный и прямой доступ к файлам

Принцип последовательного доступа следующий: когда мы открываем файл, мы можем прочитать или записать первую компоненту, затем – вторую и т.д. Для текстовых файлов это – единственный способ доступа. Например, если надо прочитать 100000-й символ, надо предварительно просмотреть все 99999 предыдущих.

Прямой способ доступа позволяет переходить к любой записи без просмотра предыдущих. Он применим только к типизированным и бестиповым файлам. Прямой доступ для типизированных файлов возможен, т.к. каждая компонента такого файла имеет определенное количество байтов, поэтому все записи можно пронумеровать, как и элементы массива.

Для перехода к нужной позиции в файле, используется процедура Seek(var f;N: longint)

f – логический файл (не текстовый), N – номер позиции. Если надо работать с x-той записью, то надо указать N=x-1.

Другие полезные процедуры:

function FilePos(var f):longint Возвращает номер текущей позиции в файле.
function FileSize(var f):longint Возвращает кол-во записей в файле

Пример 5: Позиционирование в типизированном файле.

```
type
  Etwas=record {Etwas – что-то}
    x,y:real;
    s:string[10];
  end;
var
  X:Etwas;
  f:file of Etwas;{Файл с записями типа Etwas}
  i:integer;
begin
  assign(f,'etw.dat');
  rewrite(f); {открываем для записи}
  for i:=1 to 4 do
    begin
    X.x:=random;
    X.y:=random;
    writeln('Введите строку (до 9 символов): ');
    readln(X.s);
    write(f,X); {Записываем в файл}
    end;
  close(f);
  reset(f);{Открываем для чтения}
  while not(eof(f)) do {пока не достигнут конец файла}
    begin
    read(f,X);  {считываем запись}
    writeln(X.x,' ',X.y,' ',X.s);
    end;
  seek(f,2);{Готовимся работать с 3-ей записью}
  read(f,X);{Считываем 3-ю запись}
  writeln('печатаем 3-ю запись');
  writeln(X.x,' ',X.y,' ',X.s);
  close(f);
end.
```

12.6. Буфер ввода/вывода

Прежде чем записать данные в файл или прочесть их оттуда, информация поступает в буфер – специальную область памяти, которая выделяется при открытии файла. Если надо записывать данные в файл, то информация накапливается в буфере, пока он не заполнится. Лишь затем данные записываются в файл. Считывать за 1 раз нельзя объем, превышающий размер буфера. По умолчанию размер буфера равен 128 байт; менять его размер для текстового файла можно с помощью процедуры
SetTextBuf(var f:Text;var Buf [BufSize:word]);

Buf – переменная, которая и будет служить буфером. Её тип не важен – все равно туда будут записываться char. BufSize – необязательный параметр. Если он не указан - то считается, что размер буфера равен размеру переменной Buf.

12.7. Бестиповые файлы

В типизированных файлах, независимо от типа записей, все они переводятся в машинный формат и в виде символов записываются в файл. В бестиповый файл информация тоже записывается в машинном представлении, но тип переменных, которые можно записывать в бестиповый файл может быть любым. Записываются и читаются данные из бестипового файла блоками. Размер этих блоков устанавливается при вызове процедур rewrite и reset.

Записываются данные в файл с помощью процедуры
procedure BlockWrite(var f: File; var Buf; Count: Word [; var Result: Word]);
Buf – бестиповая ссылка на переменную, от которой будут отсчитываться блоки.
Count - количество блоков, которые надо записать в файл.
Result - количество записанных в файл блоков. Этот параметр необязателен, но он помогает контролировать корректность записи информации в файл.

Для чтения данных используется процедура
procedure BlockRead(var F: File; var Buf; Count: Word [; var Result: Word]);

Пример 6: Чтение и запись данных в бестиповый файл.

```
1 : var
2 :   f:file;
3 :   i:integer;
4 :   x:integer;
5 :   A,B:array[1..10] of integer;
6 :
7 : begin
8 :   assign(f,'ohne.dat');
9 :   rewrite(f,20);{можно записывать в f блоки по 20 байт}
10:   randomize;
11:   for i:=1 to 10 do
12:     begin
13:     A[i]:=random(20);
14:     B[i]:=random(20);
15:     end;
16:   blockwrite(f,A,2);{записываем 2 блока по 20 байт, начиная с
переменной A}
17:   close(f);
18:
19:   writeln('Массив A');
20:   for i:=1 to 10 do
21:     write(A[i],' ' );
22:   writeln;
23:   writeln('Массив B');
24:   for i:=1 to 10 do
25:     write(B[i],' ' );
26:   writeln;
27:
```

```
28:   reset(f,20);{можно считывать из f блоки по 20 байт}
29:
30:   BlockRead(f,A,1);{Считываем 1-й блок}
31:   for i:=1 to 10 do
32:     write(A[i],' ' );
33:   readln;
34:
35:   BlockRead(f,A,1);{Считываем 2-й блок}
36:   for i:=1 to 10 do
37:     write(A[i],' ' );
38:   readln;
39:   close(f);
40: end.
```

В строке 9 открывается для записи бестиповый файл, причем размер 1 блока, который можно записать в файл = 20 байтам.

В строке 16 записывается в файл f 2 блока по 20 байт с помощью команды `blockwrite(f,A,2)`. Отсчет начинается от переменной A. Это означает, что первый блок будет записан в файл из переменной A, а второй – из переменной B, т.к. в памяти массив B расположен сразу после массива A. Если бы после массива A других переменных бы не было, то в файл были бы записаны данные, которые к нашей программе не имеют никакого отношения. Т.е., если поменять в объявлении переменных A и B местами, то результат программы поменяется.

Поэтому никогда не делайте так как я показал в строке 16, а вместо этого напишите 2 строки:

```
blockwrite(f,A,1)
blockwrite(f,B,1)
```

Этим вы застрахуетесь от неожиданностей.

В 28-й строке файл f открывается для чтения блоками по 20 байт. Дальше все понятно. И, наконец, последний пример:

Пример 7: Как в бестиповые файлы записываются строки.

```
1 : uses Crt;
2 : const
3 :   n=30;
4 : var
5 :   f:file;
6 :   i:integer;
7 :   s,s1,s2:string[n];
8 : begin
9 :   Clrscr;
10:   s:='Meine ';
11:   s1:='Seele';
12:
13:   assign(f,'ohnestr.dat');
14:   rewrite(f,n);{можно записывать в f блоки по n байт}
15:   blockwrite(f,s,1);
16:   blockwrite(f,s1,1);
17:   close(f);
18:
```

```
19:   reset(f,n);
20:   blockread(f,s2,1);
21:   write(s2);
22:   blockread(f,s2,1);
23:   writeln(s2);
24:   close(f);
25: end.
```

Все в этом примере аналогично примеру №6. Но посмотрите сам файл ohnestr. Вы увидите, что перед словами Meine и Seele есть по 1 символу, код которых равен числу символов в соответствующих строках. В текстовом файле этот символ не отображался бы.

12.8. О количестве обращений к файлу

Нашей целью будет подсчитать частоту встречаемости букв в тексте (т.е. в текстовом файле), написанном на английском языке. Результаты работы должны быть выведены в другой файл. Некоторые немецкие слова, используемые в программе:

schreiben – писать	die Anzahl – количество
finden – находить	die Zeit - время
der Buchstabe – буква	die Datei – файл

Пример 8: Нахождение частоты встречаемости букв в английском тексте.

```
1 : unit BuchStU;
2 :
3 : interface
4 : const
5 :   BuchAnz = 26; {количество букв в алфавите}
6 : type
7 : {длина массива в 2 раза больше кол-ва букв в алфавите,
8 : т.к. заглавные и строчные буквы считаются различными}
9 :   Alphabet=array[1..2*BuchAnz] of longint;
10:
11:   AlphInfo = record  {сам алфавит и общее количество букв}
12:     Alph:Alphabet;
13:     TotalAnzahl:longint;
14:   end;
15:
16: procedure SchrAlphInfo(var Datname:string;var A:AlphInfo);
17: procedure FindeBuchstAnz(var name:string;var A:alphInfo);far;
18: procedure FindeBuchstAnz2(var name:string;var A:alphInfo);far;
19:
20: implementation
21:
22: {печатает в файл с адресом Datname частотность}
23: procedure SchrAlphInfo(var Datname:string;var A:AlphInfo);
24: var
25:   f:text;
26:   i:integer;
27: begin
```

```
28:   assign(f,Datname);
29:   rewrite(f);
30:   for i:=1 to buchAnz do
31:     writeln(f,chr(i+96),' ',A.Alph[i]);
32:   for i:=1 to buchAnz do
33:     writeln(f,chr(i+64),' ',A.Alph[i+BuchAnz]);
34:   writeln(f,'Total Anzahl = ',A.TotalAnzahl);
35:   close(f);
36: end;
37:
38: {записывает в A частотность букв в файле с адресом name}
39: procedure FindeBuchstAnz(var name:string;var A:alphInfo);
40: var
41:   f:text;
42:   c:char;
43:   i:integer;
44: begin
45:   assign(f,name);
46:   reset(f);
47:
48:   for i:=1 to 2*BuchAnz do {Обнуляем входные данные}
49:     A.Alph[i]:=0;
50:   A.TotalAnzahl:=0;
51:
52:   while not(eof(f)) do
53:     begin
54:     read(f,c);
55:     if (ord(c)>=65) and (ord(c)<=90) then
56:       begin
57:       inc(A.Alph[ord(c)-64+BuchAnz]);
58:       continue;
59:       end;
60:     if (ord(c)>=97) and (ord(c)<=122) then
61:       inc(A.Alph[ord(c)-96]);
62:     end;
63:   for i:=1 to 2*BuchAnz do
64:     A.TotalAnzahl:=A.TotalAnzahl+A.Alph[i];
65:   close(f);
66: end;
67:
68: {записывает в A частотность букв в файле с адресом name}
69: procedure FindeBuchstAnz2(var name:string;var A:alphInfo);
70: var
71:   s:string;
72:   i:integer;
73:   f:text;
74: begin
75:   assign(f,name);
76:   reset(f);
77:
78:   for i:=1 to 2*BuchAnz do {Обнуляем входные данные}
79:     A.Alph[i]:=0;
```

```
80:   A.TotalAnzahl:=0;
81:
82:   while not(eof(f)) do
83:     begin
84:     readln(f,s);
85:     for i:=1 to length(s) do
86:       begin
87:       if (ord(s[i])>=65) and (ord(s[i])<=90) then
88:         begin
89:         inc(A.Alph[ord(s[i])-64+BuchAnz]);
90:         continue;
91:         end;
92:       if (ord(s[i])>=97) and (ord(s[i])<=122) then
93:         inc(A.Alph[ord(s[i])-96]);
94:       end;
95:     end;
96:   for i:=1 to 2*BuchAnz do
97:     A.TotalAnzahl:=A.TotalAnzahl+A.Alph[i];
98:   close(f);
99: end;
100: end.
```

-----------------------------основная программа--------------------------------

```
1 : uses Crt,Dos,BuchStU;
2 : type
3 :   BuchStFinder = procedure(var name:string;var A:alphInfo);
4 :
5 : procedure FindeAlphInfoMitZeit(F:BuchStFinder;var
Datname:string;var A:AlphInfo);
6 : var
7 :   Uhr,Min,Sek,mSek:word;
8 :   Uhr2,Min2,Sek2,mSek2:word;
9 : begin
10:   GetTime(Uhr,Min,Sek,mSek);
11:   F(Datname,A);
12:   GetTime(Uhr2,Min2,Sek2,mSek2);
13:   writeln('Начало работы ',Uhr,':',Min,':',Sek,':',mSek);
14:   writeln('Конец работы ',Uhr2,':',Min2,':',Sek2,':',mSek2);
15: end;
16:
17: var
18:   Datname:string;
19:   Name1,Name2:string;
20:   Al:AlphInfo;
21:
22: begin
23:   Datname:='plutar01.txt';
24:   writeln('Функция с посимвольным чтением');
25:   FindeAlphInfoMitZeit(FindeBuchstAnz,Datname,Al);
26:   writeln('Функция с построчным чтением');
27:   FindeAlphInfoMitZeit(FindeBuchstAnz2,Datname,Al);
28:
29:   name1:='plutAl1.txt';
```

```
30:   name2:='plutAl2.txt';
31:   SchrAlphInfo(Name1,Al);{записываем в name1}
32:   SchrAlphInfo(Name2,Al);{записываем в name2}
33:   readkey;
34: end.
```

В модуле BuchstU, кроме процедуры записи результата в файл, есть 2 процедуры, каждая из которых должна находить частотность букв в тексте. Все различие в них в том, что в одной процедуре происходит посимвольное считывание текста, а в другой – построчное. Думаю, что вы сами разберетесь в процедурах (они несложные). Главное помнить, что:
ord('A')=65, ord('Z')=90, ord('a')=97, ord('z')=122.

Смысл этой программы не только в том, чтобы узнать, какие буквы в английском языке встречаются чаще, а какие – реже, хотя это само по себе интересно. Мы заодно и узнаем, какая из процедур будет работать быстрее. Для этого мы воспользуемся модулем Dos, в котором есть процедура GetTime, с помощью которой можно узнать текущее время с точностью до миллисекунд. FindeAlphInfoMitZeit запускает процедуру поиска частотности и вычисляет, сколько она работает. Т.к. нам надо будет вызвать ее 2 раза для разных процедур поиска, то я ввел процедурный тип.

Текст выбран довольно объемный – «Избранные жизнеописания» Плутарха. В ней описаны биографии римских и греческих государственных деятелей – реальных и мифических. Все личности сгруппированы в пары, для того, чтобы четко проявлялись различия в поступках и характерах выдающихся (одних - в своем величии, других - в ничтожестве) людей эллинистического мира. Для нашей процедуры этот труд хорош тем, что он довольно объемный – почти 70000 строк, следовательно, - различие в скорости работы программ будет хорошо прослеживаться. На моем компьютере FindeBuchstAnz2 выполнила свою работу примерно в 9,5 раза быстрее, чем FindeBuchstAnz.

Вывод: надо обращаться к файлу как можно меньше раз – это значительно ускорит скорость работы программы.

Если вы помните, еще в 0-й главе было отмечено, что работа с жестким диском производится гораздо медленнее, чем с работа с оперативной памятью. Вот вы и убедились в этом на практике.

Задачи

1. В текстовом файле Planeten.txt находятся данные о планетах Солнечной Системы в следующем формате: название планеты, затем – масса планеты (в долях от массы Земли), далее – радиус планеты. Затем идут данные о следующей планете и т.д. Вы должны обработать этот файл и создать новый файл, в котором была бы дополнительная информация о плотности планет.
2. Подсчитайте частоту встречаемости букв в файле, рассматривая его не как текстовый файл, а как бестиповый. тогда вы сможете считывать за один раз большой блок данных в буфер. Тогда количество обращений к файлу будет заметно снижено. Вычислите время работы подпрограммы подсчета количества букв в зависимости от размера буфера. Статистику занесите в текстовый файл.
3. Вам дан текстовый файл в формате html. Вы должны перевести его в обычный текстовый формат (убрать все теги).

Проект 3: Эволюционно стабильные стратегии

1. Постановка задачи

От проблем политических, которые мы рассматривали в предыдущем проекте, перейдем к биологическим. Рассматривается популяция некоторого вида, состоящая из N особей. Внутри популяции идет активная конкуренция за ресурсы. Ресурсом может быть пища, территория, самки и т.д. У каждого индивида есть определенная стратегия поведения во время борьбы за ресурс, например «всегда нападай на соперника, пытайся отобрать ресурс, а если соперник будет активно защищаться, то отступай», или такая: «всегда нападай на врага и сражайся до победы, либо умри на поле брани». Поэтому всю популяцию можно разделить на несколько групп, в каждой из которой будут представители какой-либо одной из стратегий. В каждый период времени какая-то из стратегий будет эффективнее, чем остальные, поэтому будет изменяться как численность популяции, так и доля отдельных групп особей в общей численности. Это происходит потому, что представители той из стратегий, использование которой приносит большую выгоду, могут добыть больше ресурсов, а значит, и шансы их детенышей на выживание будут больше и численность этой группы будет расти. Нашей целью будет проследить за всеми изменениями в структуре популяции.

Эта задача допускает не только биологическую интерпретацию: в качестве популяции можно взять брокеров на бирже. Среди них есть агрессивные «особи», которые активно скупают большие количества акций, цены на которые могут сильно колебаться как в большую, так и в меньшую сторону; другие инвесторы следуют правилу «тише едешь – дальше будешь», третьи - скупают только дешевые акции, а четвертые - только дорогие, т. к. они, по их мнению, принадлежат истинным лидерам отрасли.

Докинз предложил еще один вариант той же задачи. Стратегии самцов и самок в борьбе друг за друга неодинаковы: есть самки, которые требуют длительного ухаживания за ними, а есть и другие, которые не прочь свить гнездышко с первым же самцом. Самцы, в свою очередь, могут быть верными в браке, либо быстро бросать свою невесту, или вести более сложную стратегию, например, проверять, требует ли самка длительного ухаживания, и если да, то бросать ее.

Как видите, задача имеет множество применений, а моделирование таких процессов действительно применяется в биологии. Одной из пионерских работ в этой области была работа Мэйнарда Смита, на которого ссылается сам Докинз.

- **Эволюционно стабильной стратегией (ЭСС)** для вида назовем такое распределение стратегий между членами популяции, что появление одного нового индивида с любой стратегией приведет к тому, что популяция сразу вернется в состояние с ЭСС.

Чтобы проиллюстрировать это понятие, рассмотрим простейший пример: в популяции есть 2 группы индивидов: Ястребы – особи, которые в процессе борьбы за ресурс будут всегда драться со своим соперником, и не будут отступать, пока не погибнут или не получат серьезную рану и Голуби – особи, которые будут только угрожать и демонстрировать свои боевые качества, а ввязываться в драку не будут. Будем считать, что все особи обладают равной физической силой, ловкостью, и другими боевыми качествами, а также, что особь, которая однажды приняла определенную стратегию, менять ее не будет.

При сделанных предположениях если Ястреб будет бороться с Ястребом, то до получения серьезной раны никто из них не отступит, если Ястреб будет конкурировать с Голубем, то Ястреб всегда выходит победителем, а Голубь сразу отступает, не ввязываясь в схватку, и не получает травм. Если конкурируют 2 Голубя, то они будут кичиться своими достоинствами, пока один из них не удалится. Так как физическая сила всех соперников, по предположению, одинакова, то в схватках Голубь – Голубь и Ястреб – Ястреб у каждого из индивидов шанс победить 50 % (или 0,5).

Теперь давайте введем ценности ресурсов (числа условны):

- Ценность захваченного ресурса = 60.
- Затраченное время = -20 (время тратится лишь на схватку)
- Тяжелая рана (или смерть) = -100

Рассмотрим сначала популяцию, состоящую лишь из Ястребов: так как соперником Ястреба может быть только другой Ястреб, то средний исход в одной битве будет $\frac{60-100}{2}-20=-40$

А если в популяции появится Голубь (например, вследствие мутации), то он, очевидно, всегда будет проигрывать, т.е. результат схватки для него = 0. Выходит, что хотя Голубю желаемого ресурса не увидеть, зато ноги целы, что уже немало. Очевидно, что количество Голубей будет расти и у них будет даже шанс на победу над другим Голубем, т.е. средний результат схватки будет выше 0.

Если же в популяции одни Голуби, то для них ожидаемый результат выгоды в результате схватки $(60/2)-20=10$ (тяжелая рана им не грозит). Но как только появляется хотя бы один Ястреб, то он будет постоянно выигрывать схватки, и, значит, средняя ценность схватки для него будет = 40 и количество Ястребов будет возрастать.

Т.е. простейший анализ показывает, что чистые стратегии не могут быть стабильными.

2. Моделирование жизни популяции

Ваша цель состоит в том, чтобы вы проанализировали жизнь популяции, в которой есть представители нескольких стратегий. В частности, вы должны будете проверить, придет ли популяция в состояние с ЭСС, или нет.

При написании сложных проектов, особенно если это – ваш первый опыт, часто не знаешь с чего начать, как представить себе структуру программы, или даже ее небольшого куска? Многие сразу начинают нервничать, так как пытаются писать все сразу, при этом программа, конечно, не работает. Так делать категорически нельзя: если вы беретесь за проект, то в первую очередь вы должны, отбросив все технические детали, воссоздать в голове модель, которая бы отражала сущность программы. Если вы это сделаете, то вы спасете себе уйму времени. Даже если вы сделаете все как надо, то в процессы программирования могут возникать самые разные проблемы, которые могут привести к изменениям структуры проекта, но если вы грамотно проведете предварительный анализ, то вам удастся сэкономить дни и даже недели.

При написании многомодульных проектов возникают дополнительные требования:

- Изменения должны легко вноситься в проект.
- Проект должен быть по возможности более универсален.
- Структура проекта должна соответствовать поставленной задаче.

Основные понятия задачи:

1. Источники дохода/потерь.
2. Стратегия особи
3. Группа особей
4. Популяция (множество групп особей).

Предположим для простоты, что физические качества членов популяции одинаковы.

Количество входных данных может быть велико, поэтому вы должны будете заносить данные о ресурсах и группах в файл.

Проектирование источников дохода/потерь

Одно из базовых – источник (die Quelle) дохода/потерь (далее – просто источник). У источника есть определенная ценность, которую мы обозначим QWert (der Wert – ценность). Ценность состоит из 2-х частей: того, сколько вы выиграете от победы и того, сколько вы проиграете в случае поражения.

Например, если один из факторов – «травма», то если вы победили (т.е. нанесли травму врагу), то ценность можно установить 30, а если вы проиграли, то травму нанесли вам, и ценность можно установить в -100.

А для учета времени неважно, победили вы в схватке, или проиграли, так что ценности и победы и поражения можно установить, скажем, в -10 (числа везде условны, и могут меняться в зависимости от исследуемого вида).

Разумеется, вы можете вместо одного источника «Травма» ввести 2:

1. Нанесение травмы
2. Получение травмы

Тогда вам не придется вводить дополнительную структуру для ценности источника. Я ее оставлю, чтобы придерживаться какого-то одного варианта, хотя вы, естественно, вольны в своем выборе.

```
QWert = record {ценность источника}
  SiegWert:real;{ценность победы}
  ErnWert:real; {ценность поражения}
end;
```

У источника есть тип: он может иметь ценность лишь в битве (0), или только в мирной схватке (1), или в любом случае (2). Например, травмы в мирной схватке не наносятся, поэтому тип источника «травма» = 1. Время тратится на любой вид схватки, поэтому его тип = 2.

Таким образом, тип Quelle можно описать так:

```
Quelle = record {Quelle - источник}
  typ:byte;{Typ - тип}
 {Typ=0 => Ресурс имеет ценность только в битве}
 {Typ=1 => Ресурс имеет ценность только в мирной схватке}
 {Typ=2 => Ресурс имеет ценность всегда}
  Name:string;
  Wert:QWert;
end;
```

Проектирование стратегий

Группа (die Gruppe) особей – это множество индивидов, обладающих одинаковой стратегией (die Strategie). Значит, чтобы описать группу, понадобится указать количество особей и стратегию. Но как задать стратегию особи? Например, как описать стратегию «Если соперник ведет себя агрессивно, то давай отпор, а если нет, то – не лезь в драку». Если описывать стратегию текстовой строкой, то надо, чтобы компьютер понял, что мы имеем в виду под каждым словом, которое записано в предложении. Т. е. мы сталкиваемся с той же проблемой, которую я описал в главе № 2, когда говорил о задании чисел с помощью предложений, - надо научить компьютер быть умным. Это слишком сложно, надо искать другие пути. Второй подход – задать некоторое множество слов, которое допустимо использовать в описании стратегий и некоторые правила постановки фраз, иными словами, - написать интерпретатор для языка описания стратегий. Такой «стратегический язык» дал бы нам довольно гибкие возможности в выдумывании стратегий, но пока что написать даже такой интерпретатор нам не под силу.

Можно, однако, сделать все еще проще – представить, что член популяции это просто «автомат», который при встрече с неприятелем находится в некотором состоянии (хочет драться, покрасоваться, помахать рогами или что-то еще), а потом в зависимости от неприятельского и своего собственного состояния может переходить в новое состояние.

В таком случае стратегия должна включать в себя некоторое начальное состояние (der Zustand), список всех возможных состояний (его можно сделать глобальным для всех стратегий), и функцию перехода, которая меняет собственное состояние.

В реализации этой идеи нам помогут функциональные типы.

```
type
{Следующая функция будет возвращать состояние (Zustand)
члена стратегии S в ответ на состояние врага
 Состояние может быть:
 0 - агрессия
 1 - мирное состояние
и т.д.
}
{ Eigenzustand – собственное состояние, Gegnerszustand – состояние неприятеля }
  StrZustand=function(EigenZustand,GegnerZustand:integer):integer;

  Strategie = record
    Name:string;{Название стратегии}
    AnfZustand:integer;{начальное состояние}
    WillSchlachten:StrZustand;{Эта функция задает саму стратегию}
  end;
```

Функция переходов, которая задает стратегию, будет принимать в качестве параметров свое предыдущее состояние и текущее состояние соперника, а возвращать свое новое состояние.

Для каждой стратегии надо писать отдельную функцию, которая бы позволила реагировать на любое действие врага.

Докинз описывает 4 простые стратегии:

- Голубь (die Taube) всегда придерживается мирной тактики
- Ястреб (der Habicht) всегда агрессивен.
- Отпорщик (отпор – die Abfuhr, отпорщик – der Abfuhrer), если враг не пытается нападать, то ведет себя мирно, а если враг агрессивен, то будет стоять до конца.
- Задира (der Streithammel) - сначала всегда нападает, а затем, если соперник дает отпор, то сразу отступает.

Примеры функций переходов для Ястреба и Отпорщика:

```
{Ястреб}
function HabichtStr(EigenZustand,GegnerZustand:integer):integer;
begin
  HabichtStr:=0;
end;

{Отпорщик}
function AbfuhrStr(EigenZustand,GegnerZustand:integer):integer;
begin
  if EigenZustand =0 then{Если уже разозлили, то - все}
    begin
    AbfuhrStr:=0;
    exit;
    end;
  if (GegnerZustand=0) then
    AbfuhrStr:=0
  else
    AbfuhrStr:=1;
end;
```

Докинз в своей книге активно критикует многих биологов, поэтому я не могу не раскритиковать его самого: давайте рассмотрим бой двух Задир. Каждый из них сначала нападает, т.е. дает отпор с точки зрения соперника. Значит, во втором этапе схватки они должны оба убежать друг от друга. Что должны делать Задиры дальше, Докинз не говорит. Но если интерпретировать их действия в виде функции, то выходит, что, так как соперник прекратил давать отпор, то идет новое нападение на ресурс. Но ведь так думают оба Задиры, значит, - снова будет битва, затем – отступление, … и эта схватка не закончится никогда (до естественной смерти), т.е. мы попадем в бесконечный цикл. Поэтому, чтобы избежать таких проблем, я к стратегии Задиры добавлю от себя, что если Задира один раз уже отступил, то дальше он уже будет придерживаться в этой схватке стратегии Голубя, т.е. только строить боевую физиономию.

Заметьте, что ранее мы использовали переменные функционального типа только в качестве передаваемых параметров, а теперь мы храним их как переменные. Это может создать определенные трудности при создании файлов, в которых хранятся стратегии: ведь на самом деле переменная функционального типа – это не функция, а лишь ссылка на место в памяти, где хранится функция, поэтому записывать ее в файл нельзя, т.к. неизвестно, где будет расположена функция в следующий раз. Вам придется записывать в файл лишь данные нефункциональных типов, а нужные значения придавать ссылкам (настраивать их на нужные функции) в отдельной процедуре.

Понадобится также процедура, которая будет имитировать схватку между особями:

```
function WirdDieSchlacht(var S1,S2:Strategie):integer;
```

Эта функция проверяет, как будет проходить стычка между представителями стратегий S1 и S2. Она возвращает число:

- 0, если состоится кровавая стычка
- 1, если будет мирная борьба
- 2, если представитель стратегии S1 прогонит представителя стратегии S2

(в этом случае считается, что произошла мирная схватка, в которой победил представитель S1)

- 3, если представитель стратегии S2 прогонит представителя стратегии S1

Проектирование групп и популяции в целом

Группа должна содержать описание стратегии и количество ее членов, а популяция – набор групп, количество групп (в массиве могут быть пустые элементы) а также общую численность популяции (это поле необязательно, однако полезно). На основе этих незатейливых рассуждений я ввел следующие типы данных:

```
type
  Gruppe = record  {группа}
    S:Strategie;    {стратегия группы}
    Anzahl:longint; {численность группы}
  end;

{Пусть Gruppen - некоторая структура данных, содержащая все группы}
  Population = record {Набор стратегий}
    Gr:Gruppen;
    PopAnzahl:longint;{Общая численность популяции}
    GrAnzahl:integer;{Количество задействованных стратегий}
  end;
```

Какой будет структура Gruppen выбирать вам. Наверное, имеет смысл сделать ее списком (см. главу 13). В таком случае вы добьетесь большей общности.

Возможно, имеет смысл ввести еще структуру для описания всего жизненного цикла популяции, в которой должны быть функции рождаемости и смертности в популяции. Они могут в популяциях разных видов быть различны, поэтому логично и тут воспользоваться процедурными типами.

Надеюсь, что мое описание хоть немного поможет вам в вашем проекте.

Глава 13: Динамическая память

13.1. Структура памяти, доступной программе

Оперативная память, доступная программе, может быть разбита на 3 части: статическую, динамическую и стек.

- Часть ОП, в которой находится программа (и глобальные данные), называется статической.
- Стек – часть ОП, в которой хранятся локальные переменные, объявляемые в подпрограммах.
- Динамическая память (ДП) – часть ОП, которая выделяется программе и в которой программа во время выполнения может размещать переменные и удалять их оттуда, если они больше не нужны.

Размещаются эти области в ОП следующим образом:

Рис 13.1. Структура памяти, доступной программе

Статическая память находится в нижних адресах, за ней – куча (так часто называют ДП). Стек находится в верхних адресах. Он может расти вниз, а куча - вверх.

Глобальные переменные, используемые в программе, являются ее неотъемлемой частью, поэтому программа после компиляции содержит их вместе с набором команд. Т.е. чем больше размер глобальных данных, которые вы используете, тем больше места ваша программа будет занимать места на диске и, что еще важнее, в оперативной памяти (ОП).

Для того чтобы можно было эффективно управлять ДП, есть специальная программа, которая называется администратором кучи. Когда мы хотим выделить участок в динамической памяти, администратор кучи отыскивает свободный участок памяти, в котором хватило бы места для удовлетворения запроса и затем резервирует это место под данные, которые мы хотим там разместить. Когда надо освободить участок памяти, администратор кучи начинает считать его свободным (т.е. эта память может выделяться под другие переменные).

Память, которая выделена программе – это непрерывный сегмент ОП. Каждый байт обладает своим порядковым номером, следовательно можно ввести адресацию

ДП. Программы на TP выполняются под управлением операционной системы MS-DOS, поэтому мы будем пользоваться адресацией, которая используется в этой ОС. Изучая Delphi, мы рассмотрим адресацию в Windows.

- Адрес переменной складывается из 2-х частей: номера сегмента (англ. segment) и смещения (англ. offset).
- Сегмент – область памяти размером 64 Кбайт, которая начинается с адреса, кратного 16.
- Смещение – кол-во байтов, которые надо отступить от начала сегмента, чтобы указывать точно на переменную.

В памяти адрес можно трактовать как 2 числа типа word.

адрес = | Номер сегмента | смещение |

Сегмент, начинающийся с 0-го байта имеет 0-й номер, с 16-го байта – 1-й номер, с 32-го байта – 2-й номер и т.д..

Значение адреса равно 16*(номер сегмента)+смещение.

Получить доступ ко всем переменным, которые мы использовали в наших программах, можно было с помощью имени, которое служило адресом для этих переменных, точнее, компилятор на этапе компоновки имя каждой переменной заменял на ее адрес. Поэтому если мы хотим создать переменную в ДП, мы должны знать адрес этой переменной, чтобы обращаться к ней. Для этого служат указатели.

- Указатель – переменная, которая в качестве значения содержит адрес определенной области памяти.
  Указатели в TP бывают 2-х типов:
- типизированные, т.е. указатели на переменную определенного типа.
- нетипизированные (бестиповые), которые могут указывать на любой байт памяти.

Объявление указателя:

```
var
  p:pointer; {бестиповый указатель}
  pr:^Real; {указатель на переменную типа real}
```

Рассмотрим на примере основные операции с указателями:

Пример 1: Операции с указателями.

```
1 : type
2 :   Preal=^Real;{^Real - указатель на real}
3 :   PInt=^Integer; {PInt - указатель на integer }
4 : var
5 :   p:pointer;{бестиповый указатель}
6 :   pr:PReal;
7 :   z:PInt;
8 :   r:real;
9 :   x:integer;
10: begin
11:   r:=8.88;
12:   pr:=@r;{Записываем в pr адрес r }
13:   writeln('Значение переменной, на кот. ссылается pr ',pr^);
```

```
14:   p:=pr; {В переменную p можно записывать указатели любых типов}
15:   writeln(Preal(p)^);
16:
17:   x:=123;
18:   z:=@x;{Записываем в pr адрес r }
19:   writeln('Значение переменной, на кот. ссылается z ',z^);
20:   p:=z; {В переменную p можно записывать указатели любых типов}
21:   writeln(PInt(p)^)
22: end.
```

- Операция @ - операция взятия адреса.

@x – возвращает адрес переменной x.

- Получить доступ к переменной, на которую ссылается указатель можно с помощью операции разыменования - шапки после називания переменной (см. строки 13, 19).

Типизированные указатели разных типов присваивать друг другу нельзя, но тип pointer совместим с любым другим указательным типом (строки 14, 20). Операция разыменования не может быть применена к бестиповым указателям, т.к. тип переменной, на который они ссылаются неизвестен. Поэтому бестиповые указатели нужно приводить к указателю на какой-то конкретный тип данных (строки 15,21).

Универсальность бестиповых указателей делает их незаменимыми при написании подпрограмм, входные данные которых могут быть разных типов. Однако их использование усложняет программный код.

В примере 1 мы использовали указатели, чтобы ссылаться на переменные, расположенные в статической памяти. Сейчас мы разберемся, как выделять под переменные место в ДП.

Выделить память под типизированный указатель можно процедурой
New(p:pointer).

Само словосочетание «выделить память под указатель» означает не то, что надо выделить память для хранения самого указателя – он и так уже существует в памяти (неважно, в статической, динамической или стеке), - оно означает, что администратор кучи должен выделить память под переменную, на которую будет ссылаться указатель, и затем вернуть адрес этой переменной в этот самый указатель.

Несмотря на то, что параметр процедуры типа pointer, он должен представлять собой типизированный указатель (если указатель бестиповый, то ошибки не произойдет, но память под него не будет выделена). Освободить память под типизированный указатель можно процедурой
Dispose(P:Pointer).

Пример 2: Выделение и освобождение памяти (для типизированных указателей).

```
1 : const
2 :   n=5;
3 : type
4 :   Mas=array[1..n] of integer;
5 :   PMas=^Mas; {Указатель на массив}
6 :
7 : function SchaffRandMass:PMas; {создает массив случайных чисел и}
8 : var                          {возвращает указатель на него}
9 :   PM:PMas;
```

```
10:   i:integer;
11: begin
12:   New(PM); {Выделяем память под указатель}
13:   for i:=1 to n do
14:     PM^[i]:=random(100);
15:   SchaffRandMass:=PM;
16: end;
17:
18: procedure SchrMass(p:Pmas);  {Печатает массив целых чисел}
19: var
20:   i:integer;
21: begin
22:   for i:=1 to n do
23:     write(P^[i], ' ');
24: end;
25:
26: var
27:   Ms:PMas;
28: begin
29:   randomize;
30:   Ms:=SchaffRandMass;{Ms присваиваем указатель на массив}
31:   SchrMass(Ms);
32:   Dispose(Ms); {Удаляем массив из дин. памяти}
33:   readln;
34: end.
```

В функции SchaffRandMass объявлена локальная переменная PM, которая является указателем на массив. Под нее в строке 12 выделяется память, в строках 13-14 массив, на который ссылается PM, заполняется случайными числами, и указатель возвращается в качестве результата.

Замечу, что указатель на любую переменную, и на массив в частности, указывает на 1-й байт, занимаемый этой переменной в памяти. Поэтому разыменованный указатель на массив и имя массива – это, по сути, одно и то же.

Вы помните, что результат функций в TP не может быть структурированного типа, что не очень удобно. Но, как видите, мы можем возвращать не сам массив, а указатель на него, который является простым (т.е. не структурированным) типом данных.

13.2. Использование бестиповых указателей

Выделять и освобождать память под бестиповые указатели можно с помощью следующих процедур:

GetMem(p:pointer; Umf:Longint) выделяет под указатель p Umf байт (der Umfang - объем) памяти (p при этом указывает на первый из этих байтов).

FreeMem(p:pointer; Umf:Longint) возвращает в кучу Umf байт памяти, связанных с указателем p.

Еще могут понадобиться такие подпрограммы:

function Seg(x):word – возвращает номер сегмента, где расположена переменная x.

function Ofs(x):word – возвращает смещение переменной x от начала сегмента.

function MemAvail:longint – возвращает объем свободной памяти (в байтах).
function sizeof(x):integer. Аргументом может быть или переменная, или название типа. Результат – объем памяти, занимаемый переменной (или любой переменной заданного типа).

Вся свободная память обычно не представляет один большой кусок идущих подряд байтов, т.к. свободная память может быть разбита на несколько сегментов. Объем памяти, который можно выделить за один вызов, не превышает длину максимального свободного сегмента.

function MaxAvail:longint – возвращает длину максимального непрерывного куска свободной ДП.
function Ptr(seg,ofs: word):pointer – возвращает указатель на место в памяти, которое задается сегментом и смещением.

В следующем примере мы рассмотрим, как можно имитировать работу с массивом с помощью бестиповых указателей.

Пример 3: Работа с бестиповыми указателями и некоторые вспомогательные функции.

```
1 : uses crt;
2 : const
3 :   n=10;
4 : type
5 :   PInt=^Integer;
6 : var
7 :   p:pointer;
8 :   U,i:integer;
9 : begin
10:   Clrscr;
11:   writeln('Свободная память в начале работы программы
',MemAvail);
12:   U:=sizeof(Integer); {размер переменной типа integer}
13:   GetMem(p,n*U); {выделяем память под n целых чисел}
14:   writeln('Свободная память после выделения памяти под p
',MemAvail);
15:   writeln('seg(p^) = ',seg(p^),' ofs(p^) = ',ofs(p^));
16:   for i:=1 to n do
17:     PInt(longint(p)+(i-1)*U)^:=i;
18:
19:   for i:=1 to n do
20:     write(PInt(ptr(seg(p^),ofs(p^)+(i-1)*U))^,' ');
21:   writeln;
22:   freeMem(p,n*U);
23:   writeln('Свободная память после возврата памяти в кучу
',MemAvail);
24: end.
```

Думаю, что эта короткая программа требует, тем не менее, длинных пояснений.

Итак, в строке 13 выделяется 2*n байт памяти под указатель p. Т.к. n=10, то по идее объем свободной памяти должен уменьшится на 20 байт. Но не тут то было! Если

вы посмотрите на результаты работы программы, то вы увидите, что было потрачено 24 байта. Не удивляйтесь этому. Дело в том, что администратор кучи выделяет память лишь порциями, кратными 8 байтам, поэтому выделяется наименьшее число байт, большее либо равное количеству байт, которое надо выделить и при этом делящееся на 8. Разумеется, при выделении больших объемов памяти потеря лишних нескольких байт – не проблема.

В 15-й строке печатается номер сегмента и смещение той области памяти, на которую ссылается указатель p. Заметьте, что надо брать именно seg(p^), а не seg(p), т.к. надо получить номер сегмента переменной, на которую ссылается p, а не номер сегмента, в котором находится сам p.

Ofs(p^) сразу после выделения памяти под p равен либо 0, либо 8. Он обязан быть кратным 8, потому что администратор кучи выделяет блоки, кратные 8 байтам, а больше 8 он тоже быть не может, т.к. если он равен 16, то администратор кучи просто увеличит номер сегмента на 1, а смещение сделает равным 0. Вы это можете проверить, посмотрев результаты работы программы, написанной в четвертом примере.

В строках 16-17 примера №3 мы начинаем наделять те 2*n байт смыслом – до этого они были для нас просто доступной памятью. В этих 2-х строках мы трактуем эти 2*n байт как массив из n целых чисел, а p – как указатель на начало массива. Но если в примере №2 ПК знал, что переменная Pm – указатель на массив (см. строки 9, 15 примера №2), и мы могли обратиться к i-му элементу массива как к Pm^[i], то в текущем примере сам ПК знает лишь то, что p – бестиповый указатель, и даже просто разыменовать его не получится. Приходится идти другим путем.
Давайте расшифруем выражение `PInt(longint(p)+(i-1)*U)^` в строке 17.

Значение p – адрес первого элемента массива. Мы знаем, что p состоит на самом деле из двух чисел типа word: старшие 2 байта – это номер сегмента, а младшие 2 байта – это смещение относительно начала сегмента. Еще мы знаем, что `Ofs(p^)` – либо 0, либо 8. Это означает, что в младших двух байтах переменной p находится либо 0, либо 8.

`longint(p)` – это указатель p, интерпретированный как длинное целое число. Замечу, что `longint(p)`- это не адрес, а просто целое число, соответствующее переменной p. Но, если мы будем прибавлять к числу `longint(p)` число `(i-1)*U` (напомню, что U=2 – размер переменной типа integer), то фактически это число будет прибавляться лишь к 2-м младшим байтам числа `longint(p)`, а значит, что само число `longint(p)+(i-1)*U`, если его проинтерпретировать как указатель на целое число, будет ссылаться как раз на i-й элемент массива (т.к. на первый ссылается сам p).
`PInt(longint(p)+(i-1)*U)`- указатель на i-й элемент массива. Его можно разыменовать, т.к. Pint – типизированный указатель. Итого,
`PInt(longint(p)+(i-1)*U)^:=I` означает: i-му элементу массива присваивается число i.
Строки 19-20 печатают массив. Можно было бы обращаться к элементам массива так же, как это мы делали в строке 17, но я решил показать вам еще один способ.

Расшифруем выражение `PInt(ptr(seg(p^),ofs(p^)+(i-1)*U))^`
`ptr(seg(p^),ofs(p^)+(i-1)*U)` вернет указатель (бестиповый) на i-й элемент массива. Почему это так, должно быть ясно из приведенных выше соображений. Т.к. указатель бестиповый, то его надо привести к типу (в нашем случае – к указателю типа Pint), а затем его уже можно будет разыменовывать.

Наконец-то я закончил. Я пустился в такие пространные объяснения лишь затем, чтобы вы хорошо разобрались в том, как функционируют указатели. Быть может, вам кажется, что указатели – страшная и ужасная вещь, – но скоро вы увидите, что они необходимы для написания общих процедур и создания сложных структур данных.

Для того, чтобы вы еще раз посмотрели на работу функций ofs, seg, разберитесь самостоятельно в следующем примере.

Пример 4: ofs и seg.

```
1 : uses crt;
2 : var
3 :   p,p2,p3:pointer;
4 : begin
5 :   Clrscr;
6 :   writeln('Свободная память в начале работы программы
',MemAvail);
7 :   GetMem(p,4);
8 :   GetMem(p2,4);
9 :   GetMem(p3,4);
10:   writeln('Свободная память в начале работы программы
',MemAvail);
11:   writeln('seg(p^) = ',seg(p^),' ofs(p^) = ',ofs(p^));
12:   writeln('seg(p2^) = ',seg(p2^),' ofs(p2^) = ',ofs(p2^));
13:   writeln('seg(p3^) = ',seg(p3^),' ofs(p3^) = ',ofs(p3^));
14:   freeMem(p,4);
15:   freeMem(p2,4);
16:   freeMem(p3,4);
17:   writeln('Свободная память после возврата памяти в кучу
',MemAvail);
18: end.
```

13.3. Указатель Nil

Значением указателя может быть любой адрес. Но часто надо, чтобы указатель никуда не указывал (это может использоваться, например, чтобы знать, была ли выделена под указатель память или нет). Для этого есть специальное значение указателя – nil.

Например:

```
z:=nil;
writeln(longint(z));  {0}
```

Т.е. адрес, на который ссылается z – нулевой. Но нельзя просто написать

```
z:=0;
```

т.к. указатель совместим только с указателем. Естественно, вы можете вместо строки

```
z:=nil;
```

написать строку

```
longint(z):=0;
```

Но значительно нагляднее будет использовать указатель Nil.

13.4. Обобщенная сортировка массива

В 6-й главе мы научились сортировать массивы. Но мы могли их упорядочивать лишь в определенном порядке, для массива определенного заранее размера и элементами заранее определенного типа данных. В главе, посвященной подпрограммам, мы научились, используя процедурные и функциональные типы (фактически – ссылки на подпрограммы), сортировать массив в любом порядке, а используя открытые массивы мы добились того, что мы можем сортировать массивы любой длины. Однако одно ограничение – заранее определенный тип данных нам преодолеть не удалось.

Сейчас же мы напишем действительно общую сортировку, которая сумеет отсортировать массив любого размера, в любом порядке и для переменных любого типа. Алгоритм сортировки для нас несущественен, поэтому я взял пузырьковую как самую простую. Остальные алгоритмы реализуются аналогично.

Пример 5: Обобщенная сортировка массива.

```
1 : uses
2 :   Crt;
3 : type
4 :   PInt=^integer;
5 :   PStr=^string;
6 :   Vergleich=function(p1,p2:pointer):boolean;
7 : var
8 :   p:pointer;
9 :   n,i:longint;
10:   Umf:integer; {Umfang - объем}
11:
12: {Печатает массив целых чисел}
13: procedure SchrMass(p:pointer;Q:longint);
14: var
15:   i:integer;
16: begin
17:   for i:=0 to Q-1 do
18:     write((pint(longint(p)+2*i))^,' ');
19:   writeln;
20: end;
21:
22: {Побайтово копирует Umf байтов начиная с адреса p2
23: в Umf байтов, начиная с p1}
24: procedure Kopiere(p1,p2:pointer;Umf:longint);
25: type
26:   Pbyte=^byte;
27: var
28:   i:integer;
29: begin
30:   for i:=1 to Umf do
31:     Pbyte(longint(p1)+(i-1))^:=Pbyte(longint(p2)+(i-1))^;
32: end;
33:
34: {Обобщенная сортировка пузырьком - сортирует данные размера Umf
```

```
35: в порядке, заданном функцией Vergl}
36: procedure
GemeineSort(p:pointer;Umf,Quant:longint;Vergl:Vergleich);
37: var
38:   p1,p2:Pointer;
39:   p3:pointer;
40:   i,j:longint;
41: begin
42:   writeln('Mem in Sort',MemAvail);
43:   getMem(p3,Umf);
44:
45:   for i:=Quant-2 downto 0 do
46:     for j:=0 to i do
47:       begin
48:       p1:=pointer(longint(p)+j*Umf);
49:       p2:=pointer(longint(p)+(j+1)*Umf);
50:       if Vergl(p1,p2)=true then
51:         begin
52:         Kopiere(p3,p2,Umf);
53:         Kopiere(p2,p1,Umf);
54:         Kopiere(p1,p3,Umf);
55:         end;
56:       end;
57:
58:   freemem(p3,Umf);
59:   writeln('Mem in Sort',MemAvail);
60: end;
61:
62: {Целочисленное больше}
63: function Mehr(p1,p2:pointer):boolean;far;
64: begin
65:   Mehr:=PInt(p1)^>PInt(p2)^;
66: end;
67:
68: {Строковое больше}
69: function StringMehr(p1,p2:pointer):boolean;far;
70: begin
71:   StringMehr:=PStr(p1)^>PStr(p2)^;
72: end;
73:                  //////////основная программа//////////
74: begin
75:   Clrscr;
76:                    { Сортируем массив из целых чисел}
77:
78:   Umf:=sizeof(Integer);{Размер целой переменной}
79:   n:=40;
80:   writeln('Mem = ',Memavail);
81:   getMem(p,n*Umf); {Резервируем память под весь массив}
82:   writeln('Mem = ',Memavail);
83:   randomize;           {Заполняем элементы массива}
84:
85:   for i:=0 to n-1 do  {(он расположен в динамич. памяти)}
```

```
86:     (pint(longint(p)+Umf*i))^:=random(100);
87:
88:   writeln('исходный массив');
89:   SchrMass(p,n);
90:
91:   GemeineSort(p,Umf,n,Mehr); {Сортируем массив с ф-ей сравнения
Mehr}
92:   writeln('отсортированный массив');
93:   SchrMass(p,n);
94:
95:   freeMem(p,n*Umf);{освобождаем память}
96:   writeln('Mem = ',Memavail);
97:
98:                      {сортируем массив строк размера 30 байт}
99:   Umf:=30;
100:   n:=4;
101:   writeln('Mem = ',Memavail);
102:   getMem(p,n*Umf);{Выделяем память под 4 строки}
103:   writeln('Mem = ',Memavail);
104:   {Заполняем их данными}
105:   (pstr(longint(p)+Umf*0))^:='Heinrich';
106:   (pstr(longint(p)+Umf*1))^:='Ulrich';
107:   (pstr(longint(p)+Umf*2))^:='Anders';
108:   (pstr(longint(p)+Umf*3))^:='Harald';
109:
110:   writeln('исходный массив');
111:   for i:=0 to n-1 do
112:     write(pstr(longint(p)+Umf*i)^,' ');
113:   writeln;
114:   GemeineSort(p,Umf,n,StringMehr); {Сортируем строки}
115:
116:   writeln('отсортированный массив');
117:   for i:=0 to n-1 do
118:     write(pstr(longint(p)+Umf*i)^,' ');
119:   writeln;
120:   freeMem(p,n*Umf);{освобождаем память}
121:   writeln('Mem = ',Memavail);
122:   readkey;
123: end.
```

В строке 6 задан функциональный тип Vergleich – (vergleichen - сравнивать) - функция сравнения для двух указателей (точнее, для переменных, на которые они ссылаются).

Следующая процедура рассматривает p как указатель на массив целых чисел из Q элементов, и печатает элементы этого массива.

```
procedure SchrMass(p:pointer;Q:longint);
```

Т.к. мы не знаем тип элементов, из которого состоит массив, то надо как-то реализовать оператор присваивания. Следующая процедура – один из возможных вариантов решения этой проблемы. Она побайтово копирует Umf байтов начиная с адреса p2 в Umf байтов, начиная с p1.

```
procedure Kopiere(p1,p2:pointer;Umf:longint);
```

Можно было бы сделать иначе, попросив у пользователя передавать в процедуру GemSort процедуру копирования как параметр, но мне кажется, что этот выход из положения не очень удачен.

Теперь рассмотрим основную подпрограмму. GemeineSort (gemein – общий (а также низкий, подлый)). Эта процедура рассматривает p как указатель на начало массива, состоящего из Q элементов и размер одного элемента (в байтах) равен Umf. Функция Vergl типа Vergleich используется в условном операторе для того, чтобы знать, в каком случае надо менять элементы массива.

```
procedure GemeineSort(p:pointer;Umf,Quant:longint;Vergl:Vergleich);
```

В основной программе сначала создается в ДП массив целых чисел, который сортируется по возрастанию (функцией Mehr (больше)). Потом этот массив удаляется из ДП, и создается массив строк, который затем сортируется в алфавитном порядке.

13.5. Указатели и ссылки

Ссылки, с помощью которых можно передавать параметры в подпрограммы, также могут указывать на определенные области памяти. Отличие их от указателей в том, что их нельзя разыменовывать (они разыменовываются автоматически). Благодаря этому можно не использовать нотацию указателей. В Delphi ссылки используются гораздо чаще, чем в TP, т.к. работать с объектами можно лишь посредством ссылок.

13.6. Динамические структуры данных

На протяжении всей этой книги единственной структурой данных, которую мы с вами использовали, был массив. Матрицы, строки и множества – тоже массивы. Давайте разберемся со свойствами массивов. Рассматривать будем следующие операции: добавление элемента, удаление элемента, доступ к элементу и поиск элемента. Массивы могут быть статическими и динамическими. Статические массивы – массивы, размер которых фиксирован и не может изменяться, а динамические массивы – это массивы, которые могут изменять свой размер с течением времени.

Доступ

В массиве доступ к любому элементу можно получить за 1 операцию, т.к. элементы массива индексированы.

Удаление

Удаление элемента выполняется за n операций, где n – длина массива, т.к. каждый элемент, который находится справа от того, который надо удалить, надо сдвинуть влево.

Вставка элемента

Если массив статический, то для того, чтобы можно было вставить элемент, в нем должно быть зарезервировано дополнительное место. Тогда можно освободить место для элемента, сдвинув все элементы вправо, а затем вставить требуемый элемент. Если же места нет, то вставить элемент в массив нельзя.

Пусть массив динамический. Тогда возможны 2 случая:

1. После массива есть некоторое количество свободного места (зарезервированного). Тогда элемент можно вставить точно так же, как и в статический массив.

Память, занятая массивом	Свободное место	Другие переменные

2. Если места для размещения дополнительного элемента массива нет, то надо найти новый непрерывный участок памяти, в который поместился бы массив вместе с новым элементом. Если он есть, то надо скопировать в него все элементы, включая тот, который надо вставить в требуемом порядке и затем освободить участок памяти, который раньше занимал массив.

Память, занятая массивом	Другие переменные

Вставка элемента в массив в обоих случаях составляет cn операций, где n - длина массива. Причем выделение нового участка памяти под массив тоже может потребовать много времени (и копировать придется весь массив, а не его часть).

Поиск элемента в неотсортированном массиве составляет cn операций, а в отсортированном - $c\log n$ операций (бинарный поиск).

Итак: доступ к элементам массива очень быстрый, однако пополнять или удалять элементы – довольно накладно. Кроме того, массиву требуется непрерывный участок памяти (который надо еще найти). Расширение динамического массива также проблема.

Часто массивами вообще нельзя пользоваться. Например, когда мы с вами моделировали многочлены (глава «Массивы»), мы отметили, что для хранения многочлена $x^{1000} + x^{456}$ понадобится массив из не менее чем 1001 элемента (степени начинаются с 0).

Чтобы решать задачи, где массивы применять нельзя или их использование связано с вычислительными трудностями, надо использовать другие структуры данных. У каждой из них есть свои области применения. Умение выбирать подходящие структуры данных для решения задачи очень важно для программиста.

Изучение динамических структур данных начнем с простейшей структуры – стека.

13.7. Стеки

- Стек (Stapel) – структура данных, в которой удалять и добавлять элементы можно только из вершины стека.

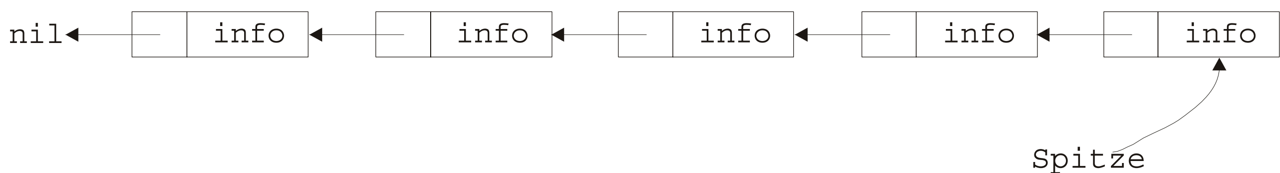
Стек используется в тех случаях, когда вам надо хранить некоторое количество элементов, а доступ вам нужен лишь к тому, который вы добавили в стек последним. Например, в главе 11 мы разбирали пример, в котором надо было проверить правильность расстановки нескольких видов скобок в строке. Алгоритм сводился к тому, что как только мы нашли закрывающую скобку, надо было проверить, не совпадает ли она с последней свободной открывающей скобкой. Если для хранения открывающих скобок использовать стек, то алгоритм был бы следующий: если мы нашли открывающую скобку, добавляем ее в стек. Если нашли закрывающую скобку,

то если она того же типа. что скобка в вершине стека, убираем из стека открывающую скобку. В противном случае скобки расставлены неверно.

Рожок автомата Калашникова – тоже стек.

Структура стека показана на рис. 13.2. Описывается такой стек следующем образом:

```
type
  ZKnote =^Knote;

  Knote = record {Узел стека}
    Folg:ZKnote;  {указатель на следующий узел (folgend -
следующий)}
    info:integer; {полезная информация}
  end;

  Stapel = record {Стек}
    Spitze:ZKnote; {Spitze - вершина (стека)}
  end;
```



Spitze – указатель на вершину стека
info – полезная информация узла стека

Рис. 13.2. Стек.

Кроме базовых операций стека – добавления элемента в стек и вытягивания элемента из него, мы напишем подпрограмму, которая будет печатать содержимое стека. Для того чтобы уничтожить стек из памяти, просто уничтожить его вершину недостаточно, т.к. вся остальная часть стека (кроме вершины) будет занимать место в динамической памяти, хотя получить к ней доступ в программе будет нельзя. Поэтому надо написать специальную процедуру (Tod – смерть), которая будет полностью уничтожать содержимое стека из ДП (для этого надо уничтожать по очереди каждый элемент, начиная с вершины стека).

Программный код достаточно хорошо прокомментирован, поэтому я думаю, что в деталях реализации вы разберетесь самостоятельно.

Пример 6: Использование стека.

```
1 : type
2 :   ZKnote =^Knote;
3 :
4 :   Knote = record {Узел стека}
5 :     Folg:ZKnote;  {указатель на следующий узел (folgend -
следующий)}
6 :     info:integer; {полезная информация}
7 :   end;
8 :
9 :   Stapel = record {Стек}
```

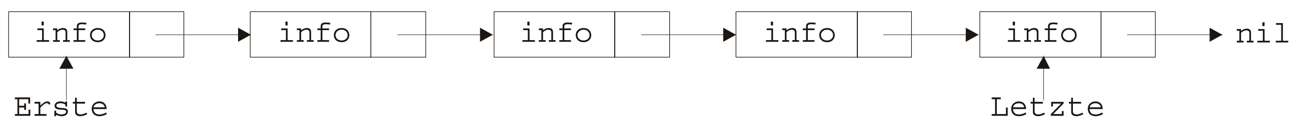
```
10:     Spitze:ZKnote; {Spitze - вершина (стека)}
11:   end;
12:
13: {добавление элемента в стек}
14: procedure Add(var S:Stapel;x:integer);
15: var
16:   z:ZKnote;
17: begin
18:   new(z);    {создаем новый узел}
19:   z^.folg:=S.Spitze; {он будет находиться перед вершиной}
20:   z^.info:=x;
21:   S.Spitze:=z;  {Делаем z вершиной стека}
22: end;
23:
24: {Возвращает true, если стек пуст}
25: function IstFrei(var S:Stapel):boolean;
26: begin
27:   Istfrei:= S.Spitze=nil;
28: end;
29:
30: {Возвращает число, записанное в вершине стека и
31: затем уничтожает вершину}
32: function Gib(var S:Stapel):integer;
33: var
34:   z:ZKnote;
35: begin
36:   if IstFrei(S)=true then
37:     begin
38:     Gib:=-1;  {если стек пуст, то возвращаем -1}
39:     exit;
40:     end;
41:
42:   z:=S.Spitze^.Folg; {запоминаем следующий за вершиной узел}
43:   Gib:=S.Spitze^.Info; {вытаскиваем информацию из вершины}
44:   dispose(S.Spitze); {уничтожаем вершину}
45:   S.Spitze:=z;   {делаем z вершиной}
46: end;
47:
48: {Печать элементов стека (начиная с головы стека)}
49: procedure Schr(var S:Stapel);
50: var
51:   z:ZKnote;
52: begin
53:   z:=S.Spitze;
54:   while z<>nil do
55:     begin
56:     write(z^.info, ' ');
57:     z:=z^.folg;  {переходим к следующему элементу стека}
58:     end;
59: end;
60:
61: {Уничтожает стек}
```

```
62: procedure Tod(var S:Stapel);
63: var
64:   z:ZKnote;
65: begin
66:   while S.Spitze<>nil do
67:     begin
68:     z:=S.Spitze;
69:     S.Spitze:=S.Spitze^.folg;
70:     dispose(z);
71:     end;
72: end;
73:
74: var
75:   S:stapel;
76:   i,x:integer;
77:
78: begin
79:   s.Spitze:=nil;
80:   writeln('MemAvail ',MemAvail);{Сколько памяти доступно в
начале программы}
81:   for i:=1 to 10 do {добавляем элемены в стек}
82:     Add(S,i);
83:
84:   write('Наш стек: ');
85:   Schr(S);  {печатем стек}
86:   writeln;
87:
88:   writeln('Извлекаем 5 элементов из стека');
89:   for i:=1 to 5 do
90:     write(Gib(S),' ');
91:   writeln;
92:
93:   write('Что осталось от стека ');
94:   Schr(S);
95:   writeln;
96:
97:   Tod(S);    {уничтожаем стек из динамической памяти}
98:   writeln('MemAvail ',MemAvail); {Проверяем, вся ли память была
возвращена в кучу}
99: end.
```

13.8. Однонаправленные списки

- Однонаправленный список – структура данных, состоящая из элементов, каждый из которых содержит помимо другой информации указатель на следующий элемент списка. Добавлять (и удалять) элементы можно в любую позицию списка.

Стек является частным случаем однонаправленного списка. Есть еще 2 структуры данных, которые являются частным случаем однонаправленного списка – очереди и деки. Реализовать основные операции с ними вам предлагается в упражнениях.



erste – указатель на первый элемент списка
letzte - указатель на последний элемент списка

Рис 13.3. Однонаправленный список

В следующем примере мы создадим однонаправленный список и реализуем все основные операции над списком, а также быструю сортировку списка. Весь программный код достаточно громоздкий, поэтому я приведу лишь несколько базовых подпрограмм с комментарием.

Структура списка очень похожа на структуру стека (только хранить надо указатель и на начало, и на конец списка).

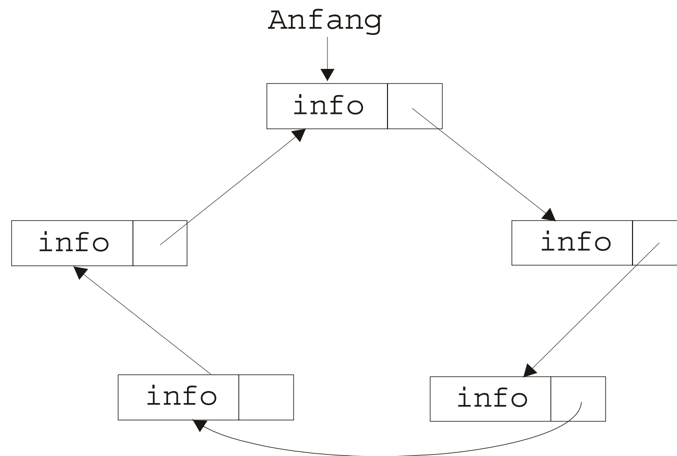
Пример 7: Работа со списком.

```
type
  lstElp=^ListeEl;
  ListeEl = record
    Nachste:lstElp; {Nachste - следующий}
    zahl:integer;   {zahl - число}
  end;
  Liste = record
    Erste,Letzte:lstElp; {erste - первый, letzte - последний}
  end;

function IstLeer(var L:Liste):boolean;
begin
  IstLeer:=L.Erste=nil;
end;

procedure ListeTod(var L:Liste); {Уничтожает список}
var
 x,y:lstElp;
begin
  with L do
    begin
    if IstLeer(L) then
      exit;
    x:=erste;  {сохраняем указатель на 1-ый элемент списка}
    erste:=nil;
    letzte:=nil;
    while (x<>nil) do {пока не дошли до конца списка}
      begin
      y:=x^.Nachste; {сохраняем указатель на следующий за x элем.}
      dispose(x);
      x:=y;       {x теперь указывает на начало списка}
      end;
    end;
end;
```

```
{Добавляет число в начало списка}
procedure StellZuAnf(var L:Liste;n:integer);
var
  x:lstElp;
begin
with L do
  begin
  new(x);
  x^.zahl:=n;
  x^.Nachste:=nil;
  if IstLeer(L) then
    begin
    Letzte:=x;
    Letzte^.Nachste:=nil;
    Erste:=Letzte;
    exit;
    end;
  x^.Nachste:=Erste;
  Erste:=x;
  end;
end;

{Добавляет число в список в заданную позицию Platz}
procedure StellZuPlatz(var L:Liste;n:integer;platz:integer);
var
  x,tmp:lstElp;
  i:longint;
begin
with L do
  begin
  if (platz=1) then {Если надо вставить элемент в начало списка}
    begin
    StellZuAnf(L,n);
    exit;
    end;
  tmp:=Erste;
  i:=1;
  while (tmp<>nil) and (i<platz-1) do{пробегаем все элементы до
(platz-1)-го}
    begin
    tmp:=tmp^.Nachste;
    inc(i);
    end;
  if (tmp=nil) then {Если в списке меньше, чем platz-1 элементов}
    exit;                {элемент вставлять не будем}
  new(x);    {Выделяем место под узел списка}
  x^.zahl:=n;
  x^.Nachste:=tmp^.Nachste; {устанавливаем ссылку на сл. эл. списка}
  tmp^.Nachste:=x; {делаем, чтобы следующим за tmp элементом был x}
  end;
end;
```

13.9. Кольцевые списки

- Кольцевым списком называется список, в котором его последний элемент ссылается на первый элемент.



Anfang – указатель на «начало» списка.

Рис 13.4. Циклический односвязный список

Иногда кольцевая структура списка помогает упростить программный код. В качестве примера разберем игру в считалочку. Игра заключается в следующем: Несколько участников становятся в круг, затем называется целое число n. После этого каждый n-ый из участников вылетает. После вылета участника отсчет начинается со следующего игрока. Победителем считается тот, кто последний останется в списке.

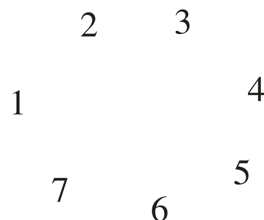


Рис. 13.5. Считалочка

Например, если играют 7 участников (см. рисунок 13.5), n=3 и начальный элемент равен 1, то вылетать они будут в таком порядке:
3, 6, 2, 7, 5, 1.

Пример 8: Считалочка.

```
1 : uses
2 :   Crt;
3 : type
4 :   ZKnoten = ^Knoten;
5 :
6 :   Knoten = record
7 :     z:integer;
8 :     Folg:ZKnoten;
9 :   end;
```

```
10:
11:   ZyklListe = record
12:     Anfang:ZKnoten;  {Anfang - начало}
13:   end;
14:
15: {Возвращает true, если пустой}
16: function IstFrei(var L:ZyklListe):boolean;
17: begin
18:   IstFrei:=L.Anfang=nil;
19: end;
20:
21: {Добавляет узел, в котором будет записано число gerz}
22: procedure AddKnoten(var Zliste:ZyklListe; zahl:integer);
23: var
24:   zeig:ZKnoten;
25: begin
26:   New(zeig);
27:   zeig^.z:=zahl;
28:   if IstFrei(ZListe)=true then
29:     begin
30:     zeig^.folg:=zeig; {один элемент должен указывать на себя же}
31:     ZListe.Anfang:=zeig;
32:     exit;
33:     end;
34:
35:   zeig^.Folg:=Zliste.anfang^.folg;
36:   Zliste.Anfang^.folg:=zeig;
37: end;
38:
39: {Печать списка}
40: procedure SchrListe(var ZListe:ZyklListe);
41: var
42:   zeig:ZKnoten;
43: begin
44:   if IstFrei(ZListe)=true then
45:     exit;
46:
47:   zeig:=ZListe.anfang^.folg;
48:   repeat
49:     write(zeig^.z,' ');
50:     zeig:=zeig^.folg;
51:   until zeig=ZListe.anfang^.folg;
52: end;
53:
54: {Освобождает память, занятую списком}
55: procedure Tod(var ZListe:ZyklListe);
56: var
57:   zeig:ZKnoten;
58: begin
59:   if ZListe.anfang=ZListe.anfang^.folg then {Если в списке один
элемент}
60:     begin
```

```
61:     dispose(ZListe.anfang);
62:     ZListe.anfang:=nil;
63:     exit;
64:     end;
65:
66:   repeat
67:     zeig:=ZListe.anfang^.folg; {Сохраняем следующий за началом}
68:     ZListe.anfang^.folg:=zeig^.folg; {Выбрасываем его из списка}
69:     dispose(zeig);   {освобождаем память, выделенную под zeig}
70:   until ZListe.anfang^.folg=ZListe.anfang;
71:   dispose(ZListe.anfang);
72:   ZListe.anfang:=nil;
73: end;
74:
75: {Считалочка, в которой выбывает каждый s-й}
76: procedure Spiel(var LZ:ZyklListe;s:integer);
77: var
78:   zeig:ZKnoten;
79:   lauf:ZKnoten;
80:   i,q:integer;
81: begin
82:   zeig:=LZ.anfang;
83:   q:=0;
84:   writeln('!!! Считалочка !!!');
85:   while zeig<>zeig^.folg do  {Пока не остался один элемент в
списке}
86:     begin
87:     for i:=1 to s-1 do  {Пробегаем s-1 человек}
88:       zeig:=zeig^.folg;
89:
90:     lauf:=zeig^.folg; {Запоминаем того, кто должен вылететь}
91:     zeig^.folg:=zeig^.folg^.folg; {выбрасываем его из списка}
92:
93:     inc(q);
94:     writeln(q,'-ым выбыл ',lauf^.z);
95:     dispose(lauf);  {удаляем из памяти того, кто выбыл}
96:     end;
97:   writeln('остался ',zeig^.z); {Пишем, кто же остался}
98:
99:   LZ.anfang:=zeig; {того, кто остался делаем началом}
100: end;
101:
102: var
103:   ZL:ZyklListe;
104:   i:integer;
105:
106: begin
107:   Clrscr;
108:   writeln('Доступная память в начале работы программы
',MemAvail);
109:   for i:=7 downto 1 do
110:     AddKnoten(ZL,i);
```

```
111:   SchrListe(ZL);
112:   writeln;
113:   writeln('Доступная память после выделения памяти под список
',MemAvail);
114:   Spiel(ZL,3);
115:
116:   Tod(ZL);
117:   writeln('Доступная память после выхода из программы
',MemAvail);
118:   readln;
119: end.
```

13.10. Характеристика списков

Точно так же, как мы охарактеризовали массивы, мы проанализируем списки.

Доступ

Добраться к произвольному элементу можно лишь за cn операций. Быстрый доступ возможен лишь к 1-му и последнему элементу списка.

Удаление и вставка элементов

Удаление элемента из середины списка требует cn операций. Но удалять элементы из начала (и конца для двунаправленных списков) списка можно за c операций (c - не зависит от длины списка). Аналогичные свойства выполняются и для вставки элемента.

Поиск

Найти элемент можно лишь за cn операций независимо от того, является ли список отсортированным или нет.

При этом списки не требуют наличия непрерывного сегмента памяти, и операции, которые требуют просмотра всего списка, например, слияние двух списков, выполняются с той же скоростью, что и аналогичные операции над массивами. Несмотря на то, что скорость вставки элементов в список линейна относительно длины списка, но величина константы значительно ниже чем при вставке элемента в массив.

Списки позволяют решить проблему, с которой мы столкнулись при моделировании многочленов – количество элементов списка, которое необходимо для представления многочлена в виде списка, равно количеству одночленов, входящих в его состав и не зависит от степени одночлена.

Фактически, преимущества массивов перед списками – скорость доступа к произвольному элементу и быстрый поиск в отсортированном массиве.

Сейчас мы с вами познакомимся еще с одной структурой данных, которая позволит быстро вставлять и искать элементы.

13.11. Деревья

Все структуры данных, которые мы рассматривали до этого, имели линейную структуру. Дерево – это одна из простейших нелинейных структур данных.

- Элемент дерева называется листом (Blatt).
- Дерево (Baum) – структура данных, состоящая из корня («главного листа»), который ссылается на несколько листьев, каждый из листьев также может ссылаться на несколько листьев и т.д.
- Дерево, у которого каждый лист ссылается на 2 других листа, называется бинарным деревом (см. рис. 13.6).

Деревья – весьма полезная структура данных. Для того чтобы опробовать их возможности, мы рассмотрим 2 примера: разбор формул с помощью деревьев и сортировку с помощью дерева.

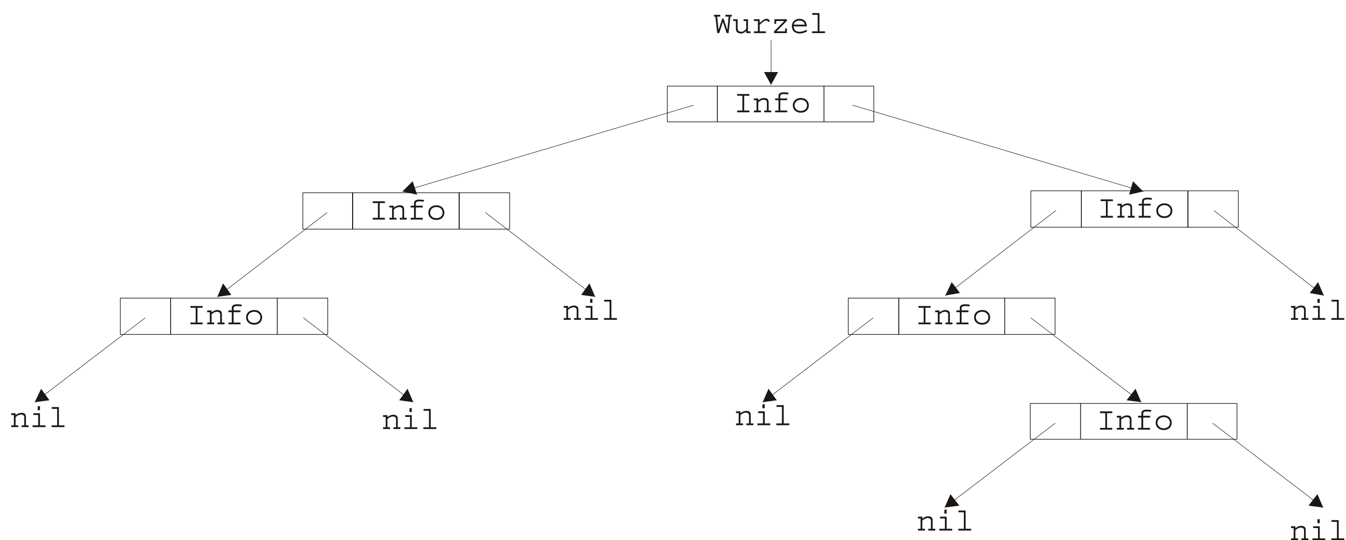


Рис 13.6. Вид бинарного дерева (Wurzel (корень) – указатель на корень дерева)

13.12. Разбор формулы

Чтобы показать сам принцип разбора формулы, и не загромождать код, мы ограничим множество допустимых формул следующими:
1. Переменные, задающиеся одной латинской буквой и константы.
2. Выражения вида $s_1 \circ s_2$, где s_1 и s_2 - формулы из пункта 1, а $\circ$ - некоторая операция из заранее заданного множества.
3. Выражения вида (выражение) операция (выражение), где:
(выражение) – любое выражение из пункта 1 или 2.
(операция) – операция из заранее заданного множества.
Например:
Допустимые выражения: x+y, (x+1)*(r+s), ((s+r)*2)+(r+(s/2))
Недопустимые выражения: (x-3)*x+3, (x+4)*(xy+4)

Целью будет построить т.н. дерево формулы.

Строится оно так: сначала выбирается самая старшая операция (у формул, которые мы разбираем, она определяется единственным образом). Если таковая есть, то она становится вершиной, а листы заполняются соответственно правым и левым подвыражением, если операций нет, значит формула является элементарной (из пункта 1), следовательно, та переменная или константа становится вершиной, а листы получают значение nil.

Например, для выражения ((s+r)*2)+(r+(s/2)) дерево формулы приведено на рис. 13.7.

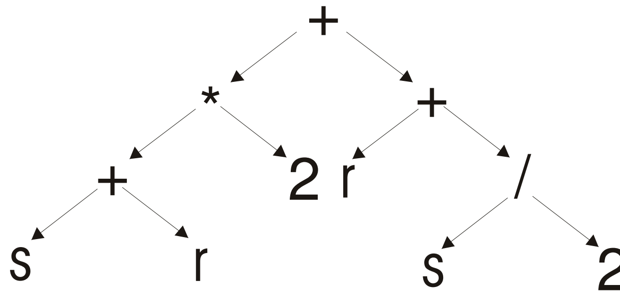


Рис 13.7. Дерево формулы для выражения ((s+r)*2)+(r+(s/2)).

После того, как дерево построено, можно выполнять с ним различные операции, например, подстановку выражений вместо переменных, сложение нескольких формул (заданных деревьями) и т.д.

Алгоритм построения дерева по строковому выражению реализован в следующем примере в подпрограмме `BaumAusZeile`.

Нам понадобится и обратный алгоритм – печать дерева формулы в виде строки:

Алгоритм - рекурсивный:

если левый лист текущего поддерева – операция, то
    печатаем ‘(‘,
    печатаем левое поддерево
    печатаем ‘)’
иначе
    печатаем левое поддерево
печатаем операцию в вершине дерева
если правый лист текущего поддерева – операция, то
    печатаем ‘(‘,
    печатаем правое поддерево
    печатаем ‘)’
иначе
    печатаем правое поддерево

Эта подпрограмма в следующем примере называется `BaumZurZeile`.

В целях экономии места исходный код следующих подпрограмм напечатан не будет:

```
{Ищет закрывающую скобку (Klammer) для скобки, индекс которой = n}
 function Finde2Klammer(const z:string;n:integer):integer;
{Проверяет, правильно ли расставлены скобки в строке}
 function IstKorrekt (const z:string):boolean;
{Ищет первый символ операции в строке}
function FindeOper(const z:string;s:charset):integer;
```

Пример 9: Распознавание формул с помощью бинарного дерева.

```
1 : const
2 :   Menge = ['/','*','-','+']; {Множество всех операций}
3 :
4 : type
5 :   Monom = record {одночлен}
6 :     koef:real;
7 :     c:char;
8 :   end;
9 :
10:   ZBlatt=^Blatt;
11:
12:   Blatt = record {лист}
13:     m:Monom;
14:     linke:ZBlatt; {левый лист}
15:     recht:ZBlatt; {правый лист}
16:   end;
17:
18:   Baum = record {дерево}
19:     Wurzel:ZBlatt;    {Корень}
20:   end;
21:
22:   charset = set of char;
23:
24:
…
108: {Формирует дерево формулы из строки}
109: function BaumAusZeile(const z:string):ZBlatt;
110: var
111:   Bl:ZBlatt;
112:   op,kl:integer;
113:   kode:integer;
114: begin
115:   new(Bl);
116:   if z[1]='(' then
117:     op:=Finde2Klammer(z,1)+1 {Где операция}
118:   else
119:     op:=FindeOper(z,Menge); {ищем первую из операций}
120:
121:   Bl^.m.c:=z[op];
122:   Bl^.m.koef:=1;
123:
124:   if z[op-1]=')' then {Если перед операцией стоит выраж. в 
скобках}
125:     BL^.linke:=BaumAusZeile(copy(z,2,op-2))
126:   else        {Строим элемент}
127:     begin
128:     new(Bl^.linke); {Выделяем место под лист}
129:     Bl^.linke^.linke:=nil;  {Надо обязательно обнулить}
130:     Bl^.linke^.recht:=nil;      {Ссылки листа}
131:
132:     if (ord('0')<=ord(z[op-1]))and(ord(z[op-1])<=ord('9')) then
133:       begin                                      {если цифра}
```

```
134:       val(copy(z,1,op-1),Bl^.linke^.m.koef,kode); {в коэффиц.
записываем все число}
135:       Bl^.linke^.m.c:='?';
136:       end
137:     else   {если переменная}
138:       begin
139:       Bl^.linke^.m.koef:=1;
140:       Bl^.linke^.m.c:=z[op-1];
141:       end;
142:     end;
143:
144:   if z[op+1]='(' then  {Если после операции стоит выраж. в
скобках}
145:     BL^.recht:=BaumAusZeile(copy(z,op+2,length(z)-(op+1)))
146:   else
147:     begin
148:     new(Bl^.recht);  {Выделяем место под лист}
149:     Bl^.recht^.linke:=nil; {Надо обязательно обнулить}
150:     Bl^.recht^.recht:=nil;      {Ссылки листа}
151:
152:
153:     if (ord('0')<=ord(z[op+1]))and(ord(z[op+1])<=ord('9')) then
154:       begin                                  {если цифра}
155:       val(copy(z,op+1,length(z)-op),Bl^.recht^.m.koef,kode); {в
коэффиц. записываем все число}
156:       Bl^.recht^.m.c:='?';
157:       end
158:     else   {если переменная}
159:       begin
160:       Bl^.recht^.m.koef:=1;
161:       Bl^.recht^.m.c:=z[op+1];
162:       end;
163:     end;
164:
165:   BaumAusZeile:=Bl;
166: end;
167:
168: {Перевод дерева формулы в строковое представление}
169: function BaumZurZeile(ZB:ZBlatt):string;
170: var
171:   tmp,tmp2:string;
172: begin
173:   if ZB=nil then
174:     begin
175:     BaumZurZeile:='';
176:     exit;
177:     end;
178:
179:   if not(ZB^.m.c in ['+','-','/','*']) then {Если в узле не
операция}
180:     if ZB^.m.c<>'?' then   {если переменная}
181:       if ZB^.m.koef=1 then {коэффициент = 1}
```

```
182:         BaumZurZeile:=ZB^.m.c
183:       else
184:         begin
185:         str(ZB^.m.koef,tmp);
186:         BaumZurZeile:=tmp+'*'+ZB^.m.c;
187:         end
188:     else                {если число}
189:       begin
190:       str(ZB^.m.koef,tmp);
191:       BaumZurZeile:=tmp;
192:       end
193:   else         {в узле операция}
194:     begin
195:     if ZB^.linke<>nil then
196:       if not(ZB^.linke^.m.c in Menge) then {не операция}
197:         tmp:=BaumZurZeile(ZB^.linke)
198:       else
199:         tmp:='('+BaumZurZeile(ZB^.linke)+')';
200:
201:     if ZB^.recht<>nil then
202:       if not(ZB^.recht^.m.c in Menge) then {не операция}
203:         tmp2:=BaumZurZeile(ZB^.recht)
204:       else
205:         tmp2:='('+BaumZurZeile(ZB^.recht)+')';
206:
207:     BaumZurZeile:=tmp+ZB^.m.c+tmp2;
208:     end;
209: end;
210:
211: procedure Totebaum(ZB:ZBlatt); {Уничтожение дерева и
возвращение памяти в кучу}
212: begin
213:
214:   if ZB^.linke<>nil then
215:     ToteBaum(ZB^.linke);
216:   if ZB^.recht<>nil then
217:     ToteBaum(ZB^.recht);
218:   dispose(ZB);
219:   ZB:=nil;
220: end;
221:
222: var
223:   z:string;
224:   B:Baum;
225:
226: begin
227:   writeln('FreiSpeicher ',MemAvail);
228:
229:   z:='(x+y)*((1+r)+r)';
230:   writeln(IstKorrekt(z));
231:
232:   B.Wurzel:=BaumAusZeile(z);  {Строим дерево}
```

```
233:   writeln(BaumZurZeile(B.Wurzel));   {Печатаем, что построили}
234:
235:   ToteBaum(B.Wurzel);    {Уничтожаем дерево}
236:   writeln('FreiSpeicher ',MemAvail);
237:   readln;
238: end.
```

13.13. Сортировка с помощью дерева

С помощью дерева можно написать простой алгоритм сортировки, который в среднем работает за время порядка $n\log n$ (n – длина исходного массива). Чтобы отсортировать массив, достаточно просто по очереди вставить все элементы массива в дерево.

Алгоритм вставки числа m такой:

Если дерево пустое, то
    вставляем число его в вершину
Иначе
    Если число m больше, чем число, которое записано в вершине, то
        Вставляем число m в правое поддерево
    Иначе
        Вставляем число m в левое поддерево

Можно доказать, что в среднем для случайных массивов длины n высота полученного дерева пропорциональна $\log n$. Отсюда и следует, что в среднем этот алгоритм отсортирует массив за порядка $n\log n$ операций.

После того, как построено дерево, можно все его элементы записать в массив с помощью алгоритма, похожего на алгоритм записи дерева формулы. Но давайте рассмотрим свойства дерева, полученного в результате работы алгоритма:

1. Добавить в дерево новый элемент так, чтобы не нарушить упорядоченности, можно в среднем за $\log n$ операций (а в массив вставка элемента требует порядка n операций).
2. Поиск элемента можно проводить в среднем за $c\log n$ операций (c - некоторая константа).

Эти свойства делают деревья такого вида (называются они деревьями поиска) полезными, в случае если требуется часто вставлять элементы в уже отсортированную последовательность (например, вставлять новые слова в словарь).

Пример 10: Построение дерева поиска.

В примере в целях экономии места не печатались процедуры для работы с массивами:
RandMass – заполнение массива случайными числами
Schrmass – печать массива
ZeroMass – заполнение массива нулями

```
1 : const
2 :   n=25;
3 :
```

```
4 : type
5 :   Mas=array[1..n] of integer;
6 :
7 :   ZBlatt = ^Blatt;
8 :
9 :   Blatt = record
10:     zahl:integer;
11:     linke,recht:ZBlatt;
12:   end;
13:
14:   Baum = record
15:     Wurzel:ZBlatt;
16:   end;
{-------------  Процедуры для работы с деревом  ------------}
46:
47: {Записывает элементы дерева в массив}
48: procedure BaumZumArr(var B:Baum;var A:array of integer);
49: var
50:   k:integer;
51:
52: procedure SchrBlatt(ZB:ZBlatt);   {Вложенная процедура}
53: begin
54:   if ZB<>nil then
55:     begin
56:     SchrBlatt(ZB^.linke); {Записываем левую ветку дерева}
57:     A[k]:=ZB^.zahl;
58:     inc(k);
59:     SchrBlatt(ZB^.recht); {Записываем правую ветку дерева}
60:     end;
61: end;
62:
63: begin
64:   k:=0;
65:   SchrBlatt(B.Wurzel);
66: end;
67:
68: {Находит для числа int место в массиве и записывает его туда}
69: procedure StellInt(int:integer;zb:ZBlatt);
70: begin
71:   if int<=zb^.zahl then
72:     if zb^.linke=nil then
73:       begin
74:       new(zb^.linke);
75:
76:       zb^.linke^.linke:=nil;
77:       zb^.linke^.recht:=nil;
78:
79:       zb^.linke^.zahl:=int;
80:       end
81:     else
82:       StellInt(int,zb^.linke)
83:   else
```

```
84:     if zb^.recht=nil then
85:       begin
86:       new(zb^.recht);
87:       zb^.recht^.linke:=nil;
88:       zb^.recht^.recht:=nil;
89:
90:       zb^.recht^.zahl:=int;
91:       end
92:     else
93:       StellInt(int,zb^.recht);
94: end;
95:
96: function BaumAusMass(var A:array of integer):ZBlatt;
97: var
98:   ZB:ZBlatt;
99:   i:integer;
100: begin
101:   new(ZB);
102:   ZB^.zahl:=A[0];
103:   zb^.linke:=nil;
104:   zb^.recht:=nil;
105:
106:   for i:=1 to high(A) do
107:     StellInt(A[i],ZB);
108:   BaumAusMass:=ZB;
109: end;
110:
111:
112: procedure Totebaum(ZB:ZBlatt); {Уничтожение дерева и
возвращение памяти в кучу}
113: begin
114:   if ZB^.linke<>nil then
115:     ToteBaum(ZB^.linke);
116:   if ZB^.recht<>nil then
117:     ToteBaum(ZB^.recht);
118:   dispose(ZB);
119:   ZB:=nil;
120: end;
121:
122:
123: var
124:   M,M2:Mas;
125:   B:Baum;
126: begin
127:   writeln;writeln;
128:   writeln('FreiSpeicher ',MemAvail);
129:
130:   RandMass(M,n);
131:   SchrMass(M);
132:   writeln;
133:
134:   B.Wurzel:=BaumAusMass(M); {Строим дерево поиска}
```

```
135:
136:   BaumZumArr(B,M2);  {Записываем содержимое дерева в массив}
137:   SchrMass(M2);
138:   writeln;
139:   ToteBaum(B.Wurzel);  {Уничтожаем дерево из кучи}
140:   writeln('FreiSpeicher ',MemAvail);
141: end.
```

Как видите, деревья могут быть очень полезными в практических задачах. Заметьте, что 3 алгоритма, которые мы использовали для сортировки, работающие в среднем за $cn\log n$ операций – сортировка вставками, слиянием, и с помощью дерева поиска – наиболее просто записываются в рекурсивной форме.

13.14. Характеристика деревьев

Доступ

Для того, чтобы определить понятие доступа к элементу дерева поиска, надо ввести индексацию. Индекс элемента можно задать, например, используя набор битов. Если на данном шаге стоит 0, то поворачиваем налево, если 1 – то направо, а если больше нет чисел, то мы нашли требуемый элемент.

Если индексация введена таким образом, то ясно, что любой элемент мы можем найти за количество операций, ≤ высоте дерева (т.е. количеству элементов в наибольшей ветке). Если дерево достаточно сбалансированно, то его высота будет порядка $\log n$, где n - количество элементов дерева.

Вставка и поиск

Операции вставки нового элемента и поиска элемента по значению выполняются быстро, т.к. если дерево заполнено плотно, то высота дерева будет порядка $\log n$, где n - количество элементов дерева. Если же дерево разрежено, то есть большой шанс того, что для нового элемента найдется место в середине дерева, и не придется просматривать много элементов.

В целом, с помощью деревьев поиска мы можем выполнять операции поиска и вставки элемента быстрее, чем аналогичные операции для списков, при этом для деревьев, как и для списков не нужен непрерывный сегмент в памяти. Но у деревьев поиска есть и недостаток – слить 2 таких дерева за линейное время не получится.

Если нужна и древовидная структура и возможность быстрого объединения структур, то можно добиться и этого (например, биномиальные кучи, фибоначчиевы кучи). Об этих структурах данных вы можете прочитать, например, в книге Кормена, Лейзерсона и Ривеста («Алгоритмы. Построение и анализ»).

Задачи

1. Создать динамический массив, элементами которого были бы бестиповые указатели. Этот массив должен быть записью, у которой кроме собственно массива будет поле с количеством элементов в массиве. Реализуйте следующие процедуры:
   - Добавление элемента в конец массива
   - Добавление элемента в заданную позицию

- Удаление элемента с заданным индексом
- Сортировка элементов массива (в обобщенном виде, естественно).

Расширение массива на один элемент может потребовать выделения для массива новое место в памяти, копирование элементов в это место и возвращение в кучу памяти, которую прежде занимал массив. Это очень накладно, если надо такую процедуру проводить много раз. Поэтому добавьте возможность установки количества элементов массива «про запас». Для этого введите дополнительно поле, в котором бы хранилось количество места в памяти, которое занимает массив.

2. Дек – частный случай однонаправленного списка, в котором разрешено добавлять и удалять элементы только с концов списка. Создайте запись «Дек» и реализуйте все основные операции с деком.
3. Очередь – частный случай однонаправленного списка, в котором разрешено добавлять элементы только в хвост очереди, а удалять – только из головы очереди. Создайте запись «Очередь» и реализуйте все операции с ней.
4. Списокназвыается двунаправленным, если у каждого элемента списка есть ссылка и на следующий, и на предыдущий элемент. Создайте запись «Двунаправленный список» и реализуйте все основные операции.
5. На основе списка реализуйте работу с одночленом от любого количества переменных. Для одночлена реализуйте следующие операции:
   - Умножение одночлена на одночлен
   - Приведение одночлена к каноническому виду (когда названия всех переменных отсортированы в алфавитном порядке).
   - Перевод одночлена в строку (например, в таком формате: 2*x*y^2*t^10).
   - Возведение одночлена в степень
   - Построение одночлена, записанного в виде строки в формате, описанном выше.
6. Представляя многочлен как список одночленов, реализуйте все основные операции с ним, а именно:
   - Сложение и вычитание многочленов
   - Приведение подобных членов
   - Приведение многочлена к каноническому виду (одночлены приведены к каноническому виду и упорядочены по убыванию степеней, при их равенстве смотрятся степени первых переменных, затем – вторых и т.д.).
   - Перевод многочлена в строку (например, в таком формате: 2*x*y^2*t^10 - 4*x*z + 6*f^2).
   - Возведение многочлена в степень (естественно, алгоритм должен работать за время, пропорциональное логарифму от степени, в которую надо возвести многочлен)
   - Построение многочлена, записанного в виде строки.
7. Реализуйте стек и очередь с помощью массива
8. Напишите функцию поиска элемента в дереве.
9. Реализуйте битовую сортировку массива, используя в качестве дополнительной памяти последовательность битов в динамической памяти.
10. Напишите следующие подпрограммы:
    - Составление полного набора слов, которые встречаются в некотором файле.

Сортировка этого набора слов по алфавиту (слов может быть очень много, поэтому сортировка должна быть эффективна).

Проект 4: Поиск сходных слов

Пусть даны 2 языка одной языковой группы (например: немецкий и английский - из германской группы). Наша цель - научить человека, который владеет одним из этих языков другому. Можно сделать просто: написать учебное руководство, которое не будет учитывать сходство языков, однако при этом снижается и темп изучения, и глубина понимания языка. Гораздо лучше воспользоваться сходствами между языками.

Сходство может проявляться по-разному:

1. Сходство правил правописания
2. Сходство слов и языковых конструкций

Давайте разберемся, в чем может проявляться сходство слов.

Простейший тип сходства слов – похожее написание или произношение и при этом по крайней мере в одном из значений эти слова должны совпадать.
Если алфавиты 2-х языков принципиально различны, то можно говорить, естественно, только о сходстве произношения. Хотя можно было бы ввести сходство написания так:

Если буква (или буквосочетание) одного алфавита произносится так же, как и некоторая буква (буквосочетание) другого алфавита, то эти 2 буквы (или буква и буквосочетание) будем считать эквивалентными. Однако это определение сводит сходство написания к сходству произношения, поэтому мне кажется, что вполне разумно в языках, построенных на различных алфавитах, говорить лишь о сходстве произношения.

Приведем некоторые простейшие примеры сходств слов в различных языках.

Немецкий и английский		Немецкий и украинский	
Немецкий	Английский	Немецкий	Украинский
beginnen	to begin	das Dach	дах
bringen	to bring	die Kreide	крейда
singen	to sing	tanzen	танцювати
das Haus	a house	die Farbe	фарба
das Feld	a field	das Schach	шахи
das Schild	a shield	die Krawatte	краватка

Для человека сходство этих слов очевидно, однако программисту надо хорошо постараться, чтобы даже такие – простейшие! - сходства мог выявлять сам компьютер. Например, слова Haus и house звучат одинаково, но пишутся с существенными различиями. А в словах das Feld ("фельт") и a field "филд" наоборот – звучание отличается больше, чем написание.

В то же время существует ряд слов, в которых определять сходство очень просто (такие, как первые 3 в списке нем/англ).

Дело в том, что в английском языке инфинитив глагола строится с помощью вспомогательной частицы to, а в немецком - добавлением в конце либо -en, либо -nen (если основа оканчивается на букву n). Слова, в которых сходство можно определить чисто формально, пользуясь лишь правилами соответствующих языков, назовем **простосходными**.

Но интереснее искать сходства на более глубоком уровне.
Например:
1. Regenbogen и rainbow означает радуга (по-немецки и по-английски соответственно).

Сходство написания и произношения у них весьма отдаленное, однако дело в другом:
по-немецки Regen – дождь, Bogen – лук, поэтому дословно можно прочитать: лук дождя.
по-английски то же самое: rain – дождь, bow – лук.
2. Бегемот по-немецки будет Nilpferd – дословно "Нильская лошадь", а по-английски бегемот – river horse – "речная лошадь". Кстати, вы видите, что одно слово может соответствовать словосочетанию.
Слова, которые состоят из подслов, которые имеют свое собственное значение, называются сложными словами. В обоих рассмотренных выше случаях мы разобрали сходство сложных слов.

Иногда сходство сложных слов переходит в простое фонетическое сходство, причем слово, которое в одном языке может распадаться на несколько простых, в другом – единое слово.
Например: морж по-немецки – Walroß (Wal – кит, Roß - конь), по-английски – walrus , причем сами слова wal и rus ничего не означают.

Анализ сходств языков - очень интересная и важная задача. Т.к. в древности племена англов и саксов жили рядом с другими германскими племенами, то и сходств в староанглийском и старонемецком языках, естественно, было больше. Например, в староанглийском были такие слова: jebedmen, fyrdmen ("люди молитвы", "конные люди").

В современном немецком:	В современном английском
молитва - Gebet	молитва – pray
лошадь – Pferd	лошадь – horse

Видно, что в этих староанглийских словах больше сходства с современными немецкими словами, нежели с английскими (кстати, слово horse (хос) чем-то похоже на немецкое Roß (росс) - конь).

Этот проект – серьезная научная проблема, поэтому тому из читателей, кто собирается заняться лингвистикой, он должен понравиться. Кстати, впереди вас ждёт еще 2 проекта, связанных с лингвистикой.

Задачи

1. Дано 2 файла, в одном из которых будут храниться немецкие, а во втором - английские слова. Вы должны найти простосходные слова, и записать их в отдельный файл.

Например: пусть даны следующие файлы:
Файл 1: Немецкие слова
beginnen
gehen
bringen
strafen
wandern

Файл 2: английские слова
to sleep
to go

to bring
to get
to begin

В результате должен получиться файл:
beginnen   to begin
bringen   to bring

2. В текстах слова, естественно, встречаются не только в инфинитиве. Поэтому вы должны уметь один и тот же глагол записывать в разных формах. Выберите любой язык и напишите программу, которая бы умела склонять глаголы по лицам.
3. Напишите программу, которая могла бы переводить глаголы из одного времени в другое (например, из будущего в прошедшее).
4. Если вы знаете 2 похожих языка, найдите 2 текста, на них написанные и попытайтесь найти в них все простосходные слова, учитывая то, что они могут находиться в разных формах.
5. Научитесь склонять существительные по падежам
6. Если вы можете строить словоформы к любым словам, то вы можете составить список различных слов в тексте с учетом словоформ.
7. Проанализировав все произведения какого-то автора, составьте его словарный запас, т.е. список слов, которые он использует в своих произведениях.

Глава 14: Выведение

14.1. Программа и модульное программирование

Теперь, когда вы уже испытали на деле принципы модульного программирования, мы еще раз рассмотрим основные понятия, изученные нами в первой части книги и постараемся более глубоко вникнуть в их суть.

Быть может, самое важное определение в этой книге – это определение программы:

- Программа – последовательность элементарных команд на машинном языке.

Поэтому между программой и данными нет существенного различия: программа – это и есть данные, но для которых есть устройство, которое может их интерпретировать.

Другое фундаментальное определение:

- Программный код – запись алгоритма на каком-то языке (не обязательно языке программирования).

Очевидно, что для машинного языка понятия программного кода и программы тождественны. Давайте теперь рассмотрим структуру программного кода в модульных языках.

В модульных языках программный код состоит из 2 частей: раздела описаний и исполняемого раздела (то, что в некоторых языках описания могут располагаться в коде вперемешку с исполняемыми операторами не имеет никакого значения). Исполняемый раздел – это последовательность исполняемых операторов. Операторы могут быть простыми – не содержать вложенных операторов или составными, которые могут включать в себя другие операторы.

Таким образом, для модульных языков программный код – это не программа (даже если был бы процессор, который мог бы выполнять прямо код TP), т.к. операторы программного кода не являются неделимыми.

Подытожим, какие есть отличительные свойства у модульных языков. В 5-й главе, еще до изучения подпрограмм, рекурсии и модулей, мы установили 2 свойства, которые присущи процедурным языкам:

1. Типизированность
2. Разделенность программного кода на уровни.

Сейчас у нас есть еще больше тому подтверждений, чем было в 5-й главе.

1. Наличие структурных операторов.
2. Локальность переменных, подпрограмм, типов и других объектов программного кода.
3. Вложенность операторов: из внутреннего оператора можно переходить только к тому, в котором он непосредственно находится (например, нельзя из цикла третьей степени вложенности перейти сразу к первому циклу).
4. Внутренние для модулей подпрограммы, типы и т.д. (объявляемые в части implementation).
5. Тип record, позволяющий строить новые типы.

Само наличие подпрограмм и модулей еще не дает основания причислять язык к модульным или процедурным языкам. В Ассемблере тоже есть подпрограммы, и библиотеки подпрограмм, которые служат той же цели, что и модули. Но типизированности в них нет, и разделенностью уровней даже не пахнет.

14.2. Простейший язык программирования

Под простейшим языком программирования мы будем понимать машинный язык, содержащий как можно меньше команд, но при этом достаточно мощный, чтобы решать любые алгоритмические задачи.

Прежде всего надо уточнить понятие алгоритма. Попытки формализации этого понятия начали предприниматься лишь в 20-м веке, хотя первые алгоритмы появились за тысячи лет до этого (например, алгоритм Евклида). Разных определений понятия алгоритма было несколько. Наиболее важное из них – машины Тьюринга - было дано английским математиком и кибернетиком Аланом Тьюрингом, который внес огромный вклад в развитие компьютерных наук. Из других определений следует отметить лямбда-исчисление Алонзо Чёрча, «машины клеточных автоматов» (или «вычисляющие пространства»), теория которых была разработана независимо Джоном фон Нейманом и Конрадом Цузе.

Мы с вами будем следовать подходу Тьюринга. Машина Тьюринга (МТ) состоит из следующих частей:

1. Бесконечная лента, на которой могут быть записаны символы
2. Считывающая головка, позволяющая прочитать символ на ленте и записать на ленту символ.
3. Блок управления, в котором находится программа, по которой действует МТ.

Рис. 14.1. Машина Тьюринга

МТ может находиться в нескольких различных состояниях, каждое из которых характеризуется специфической реакцией на данные, считывающиеся с ленты. Мы можем изначально предположить, что на ленте могут быть записаны лишь символы 0 и 1, т.к. любой другой символ можно закодировать определенной последовательностью нулей и единиц.

Все данные, которые записаны на ленте будем считать числами. Система записи натуральных чисел будет такой: ноль будет кодироваться одним символом 1, число один – двумя символами 1 и т.д. Естественно, на основе натуральных чисел можно легко построить целые и рациональные числа.

На каждом шаге МТ может считать один символ текста с ленты, записать на это место новый символ, передвинуться на один символ влево (links) или вправо (rechts) и перейти в новое состояние.

В самом начале на ленту устанавливаются начальные данные – несколько слов из единиц, разделенных нулями. По обе стороны простирается бесконечная лента из нулей. МТ стартует из нулевого состояния слева от первого единичного символа ленты.

МТ прекращает свою работу по специальной команде Halt. Будем считать, что в конце работы МТ на ленте должен остаться лишь выходной результат, а сама машина должна перейти в 0-е состояние.

Пример 1: Машина Тьюринга, прибавляющая к числу 1.

Пусть на ленте в момент начала работы МТ находится только одно число. Тогда следующая МТ прибавит к этому числу единицу.

0[0] → 0R0
0[1] → 1R1
1[0] → 1 Halt 0
1[1] → 1R1

Команда 1[0] → 1R2 читается так: Если МТ находится в первом состоянии и символ на ленте равен 0, то она пишет на ленту символ 1, передвигается на один символ вправо и переходит в состояние 2.

Реализованный алгоритм прост: МТ пробегает все единицы входного числа, дописывает единицу в конец и останавливается.

Пример 2: Машина Тьюринга, складывающая два числа x, y.

Предполагается, что числа записаны подряд и разделены одним нулем. Тогда для сложения двух чисел надо поставить символ «1» между числами и удалить один символ 1 в конце последовательности единиц (или в ее начале).

0[0] → 0R0
0[1] → 1R1
1[0] → 1R2
1[1] → 1R1
2[0] → 0L3
2[1] → 1R2
3[1] → 0 Halt 0

Аналогично, хотя и с некоторыми усилиями, можно реализовать, например, умножение двух натуральных чисел и возведение одного в степень другого. Можно с помощью МТ проверять и условия (например, четность числа).

Пример 3: МТ, которая если число четное печатает 0, а в противном случае печатает 1.

0[0] → 0R0
0[1] → 0R1
1[0] → 0 Halt 0
1[1] → 0R2
2[0] → 1 Halt 0

2[1] → 0R1

Вы видите, что несмотря на свою простоту, МТ может многое. Можно было бы множить число примеров, описав многие из известных нам алгоритмов в терминах МТ.

Опираясь на функциональность МТ, Тьюринг (и независимо от него Чёрч) высказал тезис, который известен под названием **тезиса Чёрча-Тьюринга**, который утверждает, что то, что люди подразумевают под алгоритмической процедурой – это программа для МТ, и наоборот, любая программа для МТ – это некоторый алгоритм.

Как я уже писал выше, предлагались и другие попытки формализации понятия «алгоритм», но впоследствии было доказано, что все эти формулировки эквивалентны, что подкрепило уверенность большинства ученых в справедливости тезиса Чёрча-Тьюринга.

С помощью тезиса Черча-Тьюринга можно легко установить эквивалентность итеративных и рекурсивных алгоритмов: то, что любой итерационный алгоритм можно заменить на рекурсивный мы с вами уже выяснили в главе «Рекурсия». Обратное следует из того, что машину Тьюринга, которая может сделать всё, можно реализовать как один бесконечный цикл с огромным вложенным оператором case и операцией break, а значит, рекурсия не добавляет ничего нового. Следовательно, любой алгоритм можно написать с помощью как итерационного, так и рекурсивного подхода.

14.3. Все ли могут алгоритмы?

Несмотря на то, что формализация понятия алгоритма сама по себе достойная цель, однако Тьюринг, строя МТ, метил дальше. Целью Тьюринга было решить задачу, поставленную великим немецким математиком Давидом Гильбертом (10-я проблема Гильберта), которая, не больше ни меньше, заключалась в следующем: существует ли универсальная процедура, позволяющая алгоритмическим путем находить решение любой математической задачи.

Введем понятие универсальной машины Тьюринга:

- Универсальная машина Тьюринга (УМТ) – это МТ, которая имитирует работу любой машины Тьюринга на любом входном слове.

УМТ должна получать на вход кроме входного слова для машины Тьюринга, которую она имитирует, еще и саму программу этой машины. Для этого надо перевести программу для МТ в числовое представление. Это делается точно так же, как и в машинном языке для центрального процессора: команды машинного языка кодируются числами, записанными в двоичной системе[12]. Аналогично можно ввести кодировку для команд МТ. Сама же программа – это последовательность команд. Поэтому чтобы получить числовое представление программы для МТ достаточно записать подряд представления соответствующих команд. При этом возможны определенные технические сложности, но все они разрешимы без особых проблем.

Итак, каждой МТ соответствует определенный номер (вообще говоря, он зависит от кодировки). Обозначим МТ с номером n как T_n. Заметим, что не любому натуральному числу соответствует МТ, а если и соответствует, то не всегда рабочая (например, она может никогда не останавливаться).

---

[12] Тьюринг писал свои работы в 30-х годах, поэтому на самом деле идея нумерации команд была высказана раньше появления электронных компьютеров. Впервые нумерация высказываний была предпринята австрийским математиком Куртом Гёделем, поэтому ее часто называют гёделевой нумерацией.

Результат, который получается в результате действий T_n над входным словом m (т.е. слово, которое остается на ленте после окончания работы МТ), обозначим $T_n(m)$.

Теперь, после того, как мы научились кодировать программы для МТ, можно записать выражение для универсальной МТ:

$$U(n,m) = T_n(m)$$

Если на входном слове m МТ T_n остановится, то число, которое останется на ленте, - это ответ, который выдаст МТ: $T_n(m) = p$. Если же МТ не остановится, то мы запишем это так: $T_n(m) = \dagger$.

Теперь рассмотрим «проблему останова»: существует ли МТ, которая может для любой машины Тьюринга и входного слова определить, остановится ли эта машина Тьюринга на данном входном слове.

Если вдруг окажется, что такой МТ не существует то тогда, если мы принимаем тезис Чёрча-Тьюринга, то выходит, что вполне четко сформулированную алгоритмическую проблему – проблему остановки – алгоритмически решить нельзя.

Давайте предположим, что решение задачи остановки существует. Тогда можно построить следующую МТ:

$$S(n,m) = \begin{cases} 0, T_n(m) = \dagger \\ 1, T_n(m) \neq \dagger \end{cases}$$

Если существует Машина Тьюринга $S(n,m)$, то мы можем построить и другую МТ:

$$Q(n,m) = T_n(m) \times S(n,m) = \begin{cases} T_n(m), S(n,m) = 1 \\ 0, S(n,m) = 0 \end{cases}$$

(Мы ввели операцию $\times$, которая обладает такими свойствами: $a \times 1 = a$ для любого натурального a, $\dagger \times 0 = 0$).

При $m = n$, т.е. когда на вход машины Тьюринга T_n будет подаваться ее собственный номер, получим: $Q(n,n) = T_n(n) \times S(n,n)$.

Теперь давайте рассмотрим машину Тьюринга $1 + Q(n,n)$. Такая машина существует, т.к. существует МТ $Q(n,m)$. Пусть ее номер будет равен k:

$$T_k(n) = 1 + T_n(n) \times S(n,n).$$

Давайте теперь на вход машины T_k подадим ее собственный номер. Тогда получим:

$$T_k(k) = 1 + T_k(k) \times S(k,k)$$

Но если $S(k,k) = 0$ (т.е. T_k не останавливается на входном слове k), то получим, что $\dagger = 1$, что неверно.

Если же $S(k,k) = 1$, то $T_k(k) = 1 + T_k(k)$, что также неверно. Третьего случая быть не может, поэтому мы пришли к противоречию.

Следовательно, не существует машины Тьюринга, которая могла бы определить для любого входного слова и любой МТ, останавливается ли эта машина на данном входном слове.

Таким образом, Тьюринг показал, что не существует процедуры, которая позволяет решать любую алгоритмическую задачу.

14.4. Простейший модульный язык программирования

В первом разделе этой главы мы установили отличительные черты модульных языков. Во втором разделе мы, следуя Алану Тьюрингу, построили простейший язык программирования. В этом разделе мы рассмотрим, каким должен быть простейший язык программирования, удовлетворяющий условиям модульности.
Основу простейшего модульного языка должны составить:

1. Подпрограммы (с рекурсией) и модули.
2. Оператор If
3. Типы данных: любой числовой, любой символьный, pointer, record
4. Другие операторы: оператор присваивания, операторы сравнения.
5. Приведение типов (для бестиповых указателей)

Подпрограммы и модули нужны, т.к. иначе это – вообще не модульный язык. Т.к. через рекурсию и if можно выразить циклы, то циклы можно выбросить.

Из типов данных нам нужен числовой тип и символьный (надо как-то задать соответствие между символами и их кодами) – эти два типа и будут базовыми.

Pointer нужен для того, чтобы можно было задавать последовательные структуры данных (с помощью указателей можно ввести индексацию). Record позволит строить пользовательские типы, что дает возможность структурировать типы данных. На основе перечисленных 4-х типов можно создать любой тип данных, причем он будет полностью соответствовать условиям модульности языка.

В описании, которое было дано выше, я не учитывал эффективность реализации, т.е. скорость работы программ. На уровне процессора эффективно реализованы операции с вещественными числами, поэтому хотя мы можем свести операции с плавающей точкой к целочисленной арифметике, однако вычисления станут значительно медленнее.

То же справедливо и для строк. В ассемблере есть цепочечные команды, на основе которых можно выполнять строковые операции довольно быстро.

В описанном выше простейшем модульном языке есть один неприятный момент – использование бестиповых указателей. Вроде бы бестиповый указатель – это тоже тип данных, поэтому условие типизированности языка не нарушается. Но все равно хотелось бы от них избавиться но при этом не потерять той общности, которую они нам дают.

Во второй части книги мы рассмотрим то, каким путем эту проблему пытается решить объектно-ориентированная методология программирования.

14.5. Взаимодействие с операционной системой

Вы знаете, что программа – это последовательность элементарных команд, написанных на машинном языке. Набор команд процессора позволяет ему работать с аппаратурой ПК: изменять значение любой ячейки памяти, посылать команды принтеру и монитору. Логично предположить, что программа, которую мы можем написать, может заставить процессор все это делать. Однако на каком бы языке вы не писали программу, она изначально ограничена в возможностях – и ограничивает ее операционная система (ОС).

Всеми программами, которые выполняются на ПК, руководит ОС. ОС определяют иногда как «расширенную машину», имея в виду то, что ОС предоставляет набор подпрограмм, которые позволяют программам работать с аппаратурой. Иногда говорят, что ОС - управляющий ресурсами (или «менеджер ресурсов», потому что сейчас даже уборщица – «менеджер по клинингу»), т.к. она распределяет ресурсы ПК между приложениями. Мы определим ОС так:

- ОС – программа, являющаяся прослойкой между аппаратным обеспечением и прикладными программами и предоставляющая функции, позволяющие прикладным программам работать с аппаратным обеспечением и взаимодействовать друг с другом.

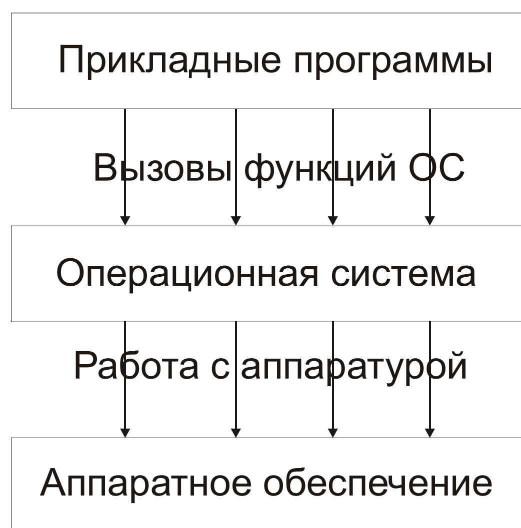


Рис 14.2. Упрощенная схема взаимодействия программ с ОС

Основные функции ОС:

1. Скрывать от прикладных программ аппаратуру и предоставлять удобный набор функций для работы с ней.
2. управление ресурсами
3. защита приложений друг от друга
4. Различные дополнительные возможности, например: поддержка многопоточности, возможность работы с несколькими процессорами и т.д.

Прежде, чем расшифровать все 4 пункта, введем важное определение:

- Процесс – выполняемая программа, включая текущие значения счетчика команд, регистров переменных, используемые ресурсы (например, открытые файлы).

Программа (последовательность команд), находящаяся на жестком диске и не запущенная на выполнение, не является процессом. Когда программа запускается, ОС выделяет ей участок динамической памяти, а также предоставляет на некоторое время доступ к процессору. При этом ОС содержит список процессов, которые находятся в памяти и информацию о них. Когда процесс отработает выделенное ему время, то он должен сохранить свои регистры, чтобы, когда ему снова дадут доступ к процессору, он мог записать во все регистры нужные значения и продолжить свою работу.

Одной программе может соответствовать несколько процессов (например, если вы открываете несколько документов с помощью Блокнота).

Можно сказать, что процесс объединяет в себе всю информацию, которая необходима программе, чтобы выполняться под управлением ОС.

Сокрытие аппаратуры

Непосредственная работа с устройствами ввода-вывода (жесткий диск (HDD – Hard Disk Drive), Flash-память и др.) довольно громоздка. Если бы программист должен был знать все тонкости работы с каждым устройством, то программирование превратилось бы в ад. Поэтому ОС содержит набор функций, которые позволяют более удобно работать с устройствами ПК. Некоторые ОС (такие, как Windows) полностью скрывают аппаратуру, поэтому работать с ней можно лишь с помощью функций ОС, а некоторые в принципе дают возможность прямого обращения к устройствам (например, MS-DOS).

Управление ресурсами

Ресурсами ПК является его аппаратура. Управление ресурсами означает, что ОС должна как можно более эффективно разделять ресурсы между процессами.

Основными ресурсами являются:

1. Процессор
2. Оперативная память (ОП)
3. Энергонезависимая память (например, жесткий диск (ЖД), флэш-ОЗУ).

Если в памяти одновременно находятся несколько процессов, то ОС должна разделить время процессора между ними.

- Часть ОС, которая распределяет время процессора между процессами, называется планировщиком процессов.

Планировщик процессов выделяет процессам квант времени. Когда он истекает, планировщик дает квант времени следующему процессу и т.д.

- Часть ОС, которая управляет дисковым пространством, называется файловой системой.
- Файл – именованная область энергонезависимой памяти.
- Каталог – системный файл, поддерживающий структуру ФС.

С точки зрения пользователя, каталог содержит набор подкаталогов и файлов.

ФС содержит информацию о каждом файле и каталоге, который находится на диске (размер, время создания, полное имя и т.д.). Сама же ОС содержит функции, которые позволяют создать новый файл на диске, открыть его для чтения и/или записи и т.д. Способ хранения файлов в разных системах может быть различен. Например, в файловых системах для CD, CD-R дисков удобно хранить файл просто как набор идущих подряд байтов. В ФС FAT-32 ЖД разбивается на сегменты одинакового размера. Каждый файл может занимать один или несколько сегментов. ФС запоминает для каждого файла начальный сегмент и в специальной FAT-таблице для каждого сегмента ЖД ФС запоминает следующий за ним сегмент.

Аналогично, в ОС есть подпрограммы для работы с другими устройствами, например, оперативной памятью (выделение динамической памяти, освобождение ее). Вы знаете, что есть менеджер кучи, который заботится о состоянии динамической памяти. Эта утилита также является частью ОС. Для того, чтобы эффективно управлять

динамической памятью, каждому процессу ОС передает некоторое количество ДП. Затем, если процессу надо больше памяти, ОС может добавить ее.

Защита приложений друг от друга

Вы знаете, что одновременно в памяти могут находиться несколько процессов, следовательно, они должны быть так разграничены, чтобы они не могли нарушать работу других приложений. Для этого ОС предоставляет каждому процессу собственное адресное пространство, в котором процесс может работать как ему вздумается. Но затем это пространство отображается на настоящую ОП.

Многопоточность

Часто надо, чтобы процесс был разбит на несколько «подпроцессов», которые действовали бы «одновременно» (т.е. псевдопараллельно). Например, если мы программируем игру в шашки, то пока мы думаем, процессор будет простаивать в ожидании хода. Гораздо эффективнее было бы часть этого времени предоставить электронному игроку, чтобы он мог думать одновременно с нами. В таком случае разбить процесс на 2 «подпроцесса» было бы разумно.

- Поток – это выполняемая программа (набор команд), включая регистры и переменные стека.

Т.е. поток – это аналог процесса, только все потоки внутри одного процесса выполняются в одном адресном пространстве (т.е. ОС память выделяет процессу в целом, а не потокам по отдельности), и совместно используют ресурсы, принадлежащие процессу.

- Если ОС может поддерживать возможность разделения процессов на несколько потоков, то такая ОС называется многопоточной.

14.6. Загрузка ОС

При включении ПК происходит процесс, который называется самозагрузкой. Центральный процессор (ЦП) построен так, что он считывает первую команду всегда из определенной области памяти. Как правило, эту область памяти конструируют так, чтобы информация, записанная на ней, не изменялась. Поэтому такую память называют ПЗУ (постоянное запоминающее устройство). В этой памяти находится специальная программа, называющаяся программой первоначальной загрузки, которая и предписывает процессору загрузить в энергозависимую память операционную систему. После того, как начинают выполняться команды ОС, она может полностью контролировать работу ПК.

Набор функций, который ОС предоставляет пользовательским программам, в Windows называется Win32 API (Application Programming Interface). Так как большую часть работы с аппаратурой ОС берет на себя, то как следствие прикладные программы становятся гораздо более аппаратно-независимыми.

Однако зачастую неудобно использовать функции ОС напрямую, поэтому во всех системах разработки приложений есть набор модулей, которые позволяют упростить выполнение тех или иных действий для программиста. Кроме того, проектировщики систем разработки хотят сделать приложения, написанные на этих программных продуктах, независимыми или практически независимыми от ОС. Изучая Delphi, мы посмотрим, насколько это им удалось.

Часть II
Объектно-ориентированное программирование

Предисловие ко 2-й части

Delphi – объектно-ориентированный язык программирования, являющийся расширением языка Pascal. Раньше этот язык программирования называли Object Pascal, а именем Delphi называли всю среду разработки приложений. В принципе, базовые возможности ООП поддерживает и Turbo-Pascal, однако набор средств, который он предоставляет, довольно скромен, поэтому TP обычно называют объектным языком.

Помимо объектно-ориентированности, важным отличием Delphi от TP является то, что среда разработки Delphi ориентирована на разработку визуальных приложений под Windows (в отличие от TP, который работал, основываясь на функциях DOS), поэтому стандартная библиотека классов и подпрограмм разработана полностью на основе функций Win32 API. Кроме того, в Delphi 2005 есть возможность программировать на платформе .NET. Следовательно, стандартных модулей, которые мы использовали в TP, в Delphi нет. Но вы не должны сильно огорчаться по этому поводу, т.к. мы не вдавались в изучение всех функций этих модулей, а использовали лишь несколько подпрограмм.

Есть аналог Delphi и для ОС Linux, который называется Kylix.

Структура следующих глав такова:

В 15-й главе мы рассмотрим основные отличия Delphi от TP, которые не касаются объектно-ориентированного программирования (ООП).

В 16-й главе мы разберемся, что такое ООП, и какие ОО-возможности есть у Delphi.

В 17-й главе вы научитесь писать оконные приложения под Windows. В частности, мы рассмотрим, что такое исключения и как с ними работать, как писать многопоточные приложения, а также изучим основные видимые компоненты. Количество встроенных компонентов огромно, и рассматривать всех их не имеет смысла, поэтому мы изучим лишь несколько самых основных, но для того, чтобы вы могли легко и быстро разбираться с другими компонентами, мы изучим свойства, которые присущи всем компонентам в целом.

В 18-й главе мы рассмотрим базовые графические возможности Delphi и научимся рисовать фракталы.

И, наконец, в последней главе мы подведем итог под всеми нашими знаниями программирования, критически разберем ООП вообще и некоторые ОО-языки в частности, а также попытаемся решить проблемы ООП, разработав новую концепцию – *везенспрограммирование (Wesensprogrammierung).*

Глава 15: Основы Delphi

15.1. Среда Delphi

Для работы вам понадобится среда разработки Delphi (например, Delphi 2005, хотя вполне подойдет и Delphi версий 6,7). Основное отличие Delphi 2005 от Delphi предыдущих версий - это поддержка программирования как под Win32, так и на платформе .NET. Наша цель – изучить объектно-ориентированное программирование, а не заниматься различными «технологиями», поэтому мы не будем рассматривать программирование на платформе .NET (ей посвящено достаточно много литературы). Основные отличия в языке между Delphi 2005 и Delphi 7 мы опишем в разделе 16.20. В остальной части учебника все примеры будут работать как под Delphi 2005, так и под Delphi версий 6,7.

Запустите Delphi 2005. После этого зайдите в пункт меню File и выберите раздел New -> VCL Forms Application – Delphi for Win32. После этого вы увидите следующее окно:

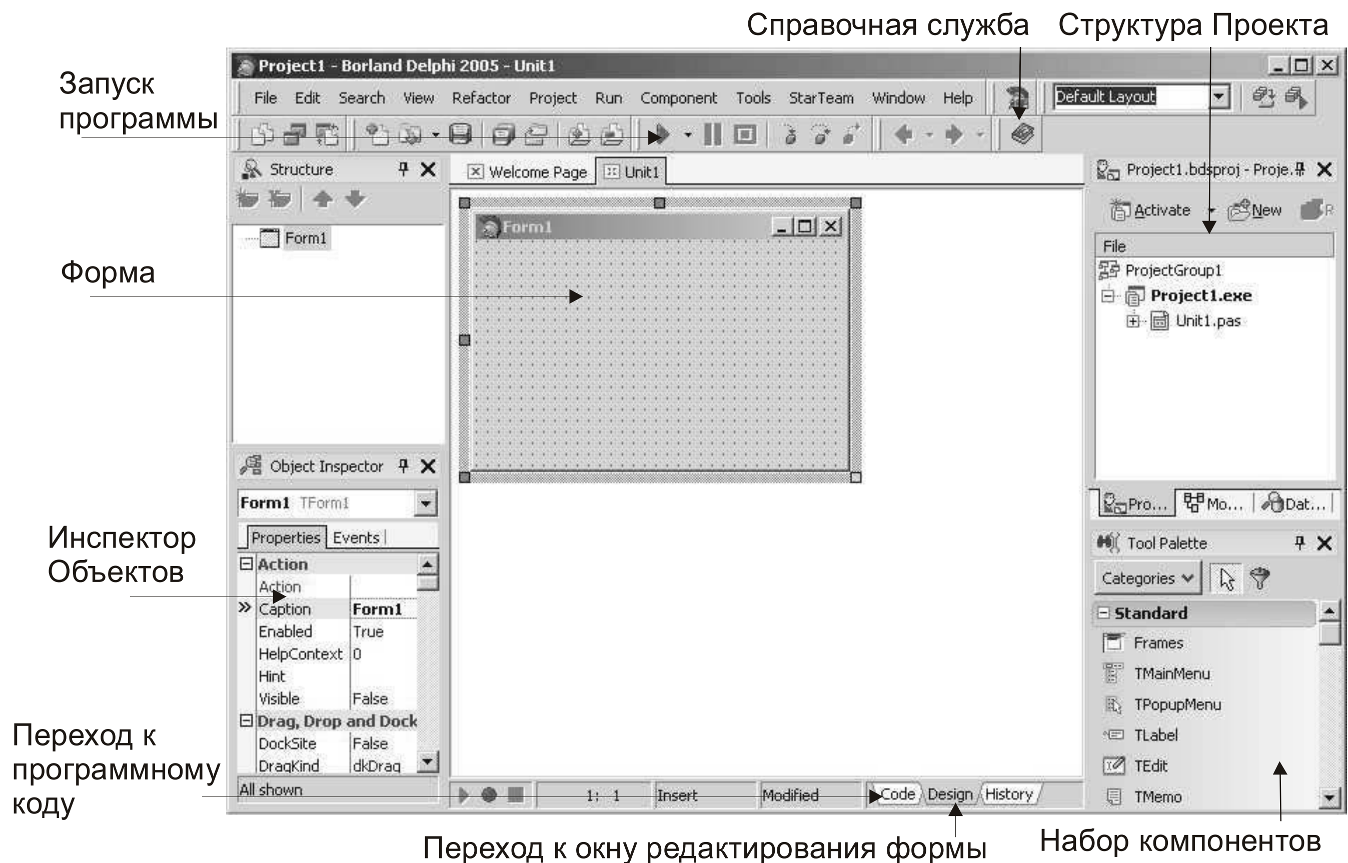


Рис 15.1. Среда разработки Delphi 2005

Краткие пояснения

- Любые оконные приложения содержат хотя бы 1 форму (так называется в Delphi окно).
- Каждая программа содержит основные объекты. Они отображаются в «Дереве объектов».

- Инспектор объектов позволяет задавать свойства объектов, которыми они будут обладать в момент загрузки.
- Кнопка Code позволяет перейти непосредственно к программному коду вашей программы. Если вы уже сделали это, то увидели, что Delphi по умолчанию генерирует стандартный код.
- Набор компонентов – это готовые примитивы, которые вы можете использовать внутри формы, например: кнопки, поля для ввода текста, различные диалоги, области рисования и т.д.
- Справочная служба - непременный помощник в вашей работе. Она довольно содержательна и при этом относительно компактна.
- Кнопка запуска позволяет запустить приложение. Также это можно сделать, нажав F9. Если вы хотите откомпилировать программу, не запуская ее, нажмите Ctrl+F9.

Для того чтобы понять, как функционирует оконное приложение, надо знать ООП. Поэтому в первых двух главах, посвященных Delphi, мы будем продолжать писать консольные приложения (т.е. приложения, которые используют окно с командной строкой).

В дальнейшем изложение будет ориентировано на студию 2005 как более новую, однако я думаю, что у читателя не возникнут большие затруднения при работе со студиями более ранних версий. Перевод приложений из Delphi 6, 7 в Delphi 2005 не сложен, однако могут быть проблемы с русскими символами: несмотря на то, что в Delphi 2005 можно использовать кириллицу даже в идентификаторах (равно как и немецкие буквы ö, ä, ü, ß), однако по непонятным причинам Delphi 2005 не разрешает писать в словах строчную букву «я». Поэтому в примерах, реализованных в Delphi 2005 я строчные «я» заменил на заглавные.

15.2. Консольное приложение в Delphi

Для того чтобы создать консольное приложение, зайдите в пункт меню File ->New->Other.
В окошке, которое вы после этого увидите, выберите Console Application в разделе Delphi Projects.
После этого вы увидите окно с простым кодом:

Пример 1: Консольное приложение.
```
program Project1;
{$APPTYPE CONSOLE}
uses
  SysUtils;
begin
  { TODO -oUser -cConsole Main : Insert code here }
end.
```

Директива компилятору {$APPTYPE CONSOLE} предназначена для того, чтобы компилятор знал, что приложение будет консольным. Тогда компилятор создаст консоль, в которой можно будет работать в командной строке. В противном случае консоль не будет создана и процедуры writeln, readln будут работать лишь для работы с файлами. Но не забывайте, что несмотря на внешний вид приложения, в TP вы

работали под управлением DOS, а консоль в Delphi – это приложение, написанное под 32-разрядные версии Windows[13].

Модуль SysUtils по умолчанию указывается в директиве uses для всех приложений автоматически, но для простейших программ он не обязателен. Никаких форм с консольным приложением не связано.

15.3. Изменения в Delphi (по сравнению с TP)

Числовые типы

В Delphi некоторые из числовых типов данных, которые мы использовали в TP, изменили свои значения, появилось много новых стандартных типов. Некоторые типы, как, например, extended, определяются в Win32 и .NET по-разному (за подробностями обращайтесь к справочной системе). Мы сейчас быстро обсудим основные отличия.

Целые типы

Основными типами являются Integer и Cardinal.

Тип	Диапазон значений	Объем занимаемой памяти
Integer	–2147483648..2147483647	4 байта
Cardinal	0..4294967295	4 байта

Все паскалевские целые типы, кроме Integer, перешли в Delphi без изменений. Появились и новые типы. Подробнее смотрите справку Delphi (ключевые слова – simple types, integer types).

Вещественные типы

Все паскалевские типы, кроме real, не изменились. Real стал эквивалентен старому типу Double. Появилось 2 новых типа: Real48 и Currency.

Тип	Диапазон значений	Значащие цифры	Объем занимаемой памяти
Real	5.0*10^–324 .. 1.7 *10^308	15-16	8
Real48	2.9*10^–39 .. 1.7*10^38	11-12	6
Currency	–922337203685477.5808.. 922337203685477.5807	19-20	8

В справочной системе можете набрать simple types, или real types, чтобы узнать более подробную информацию о вещественных типах.

Комментарий

Помимо комментария {}, который был в TP, и (* *), который остался в наследство от обычного Pascal, в Delphi перекочевал из C++ однострочный комментарий:

```
// Это - комментарий.
```

---

[13] Windows 2000 и более поздние версии Windows могут эмулировать работу DOS, поэтому многие приложения, написанные под DOS, могут работать и под этими ОС.

Использование подпрограмм

В TP результат функции мог быть только простого типа (число, символ, указатель или строка). В Delphi это ограничение снято. Результат функции может быть любого типа. В TP результат функции присваивался ее имени, а в Delphi введена специальная переменная Result, в которую надо присваивать значение функции, хотя старый способ оставлен для совместимости с более ранними версиями Паскаля.

Перегрузка подпрограмм

Перегрузкой подпрограмм называется объявление 2 подпрограмм с одинаковыми названиями, но разными сигнатурами (т.е. набором параметров и результатом).
Например, вы можете объявить 2 такие функции:

```
function MeinMax(x,y:integer):integer;overload
begin
  if x>y then
    result:=x
  else
    result:=y;
end;

function MeinMax(x,y:real):real;overload
begin
  if x>y then
    result:=x
  else
    result:=y;
end;
```

Для того, чтобы компилятор понял, что подпрограммы перегружены, после их объявления надо ставить директивы overload (т.е. перегруженная). В TP перегрузки не было, хотя она очень удобна.

Out-параметры

Появился новый тип параметров – out-параметры. Пример процедуры с out-параметром:

```
procedure Proc(out X:real;a:integer);
begin
  X:=a/2;
end;
```

Out-параметры – это ссылки, как и const- и var-параметры. Out-параметры – противоположность const-параметров: если вторые нельзя изменять, но можно использовать, то первые нельзя использовать, но можно изменять.

Умалчиваемые параметры

В Delphi можно писать функции, в которых некоторые параметры можно пропускать.
Например:

```
procedure Schweiger(a:real;b:integer=0);
```

В таком случае вы можете записать Schweiger(2.3), и компилятор поймет, что второй параметр просто равен 0. Естественно, что вызвать процедуру можно и с 2-мя параметрами, например: Schweiger(2.4, 5).

Процедуры с большим числом параметров, чем 1, писать можно, однако удобств в этом меньше, чем проблем: если вы хотите переопределить последний параметр, то надо обязательно явно указать все предыдущие. Кроме того, по умолчанию параметры могут передаваться либо как параметры-константы, либо как параметры-значения.

Соглашения по вызову функций

После заголовка функции может стоять директива, которая задает способ вызова функции. Этот способ определяет, каким образом будут передаваться параметры (через стек или через регистры), порядок передачи параметров (справа налево или слева направо) и влияет на обработку исключений.

Всего таких директив 5: **pascal, register, cdecl, stdcall, safecall**.

С помощью директивы **register** до трех параметров подпрограммы может передаваться через регистры. Остальные будут передаваться через стек. Эта директива используется по умолчанию, как наиболее эффективная.

Директива **pascal** оставлена лишь для совместимости с предыдущими версиями.

Остальные директивы используются для работы с внешними библиотеками подпрограмм и для написания собственных библиотек.

Директивы **near, far, extern** были разработаны для приложений под DOS. В программировании Win32-приложений они не влияют на работу функции и оставлены лишь для обратной совместимости.

Директива inline

При вызове подпрограмм на передачу параметров и сохранение адреса подпрограммы тратится дополнительное время. Иногда имеет смысл сделать так, чтобы функция сама встраивалась в код (т.е. вместо вызова подпрограммы все ее тело переносится в то место, откуда она должна быть вызвана). В таком случае программа станет работать несколько быстрее, однако скомпилированный код будет занимать больше места. Для того чтобы объявить встраиваемую подпрограмму, используется директива inline. Например:

```
function InFunk(x:integer):integer;inline;
begin
  result:=x*x;
end;
```

Вообще говоря, даже если программист пишет директиву inline, это не означает, что компилятор сделает ее встраиваемой. Эта директива скорее рекомендация, а не приказ. Те случаи, когда компилятор проигнорирует директиву inline, вы можете найти в документации.

Динамические массивы

Динамические массивы были введены в Delphi 4. Для того, чтобы выделить память под n элементов массива, надо вызвать процедуру SetLength(A,n), где A – название массива.

В динамическом массиве элементы нумеруются, начиная с 0.

Пример 2: Демонстрация динамических массивов.

```
program DynMass;

{$APPTYPE CONSOLE}

var
  A:Array of integer;
  i:integer;
begin
  SetLength(A,6);
  for i:=0 to 5 do
    A[i]:=1;
  for i:=0 to 5 do
    writeln(A[i]);
  readln;
end.
```

Процедуру SetLength можено применять к динамическому массиву многократно. После каждого нового вызова место, которое было занято под массив, возвращается в кучу, а вместо этого выделяется новое место в памяти.

Можно создавать и многомерные динамические массивы, причем не обязательно прямоугольные, например:

```
var
  A:array of array of real;
for i:=0 to n-1 do
SetLength(A[i],i);
```

В результате получим треугольный массив. Такие массивы могут использоваться, например, для хранения симметрических матриц (матриц, которые симметричны относительно главной диагонали).

Открытые константные массивы

В Delphi можно кроме обычных открытых массивов передавать еще и массивы констант, например:

```
function Funk(A:array of const):integer;
```

В качестве array of const можно использовать массив любых констант (даже разных типов), например:

```
Funk([1,’f’,45.22,’asadf’]);
```

Выход за границы элементов массива

Если вы в примере 2 выделите память лишь под 2 элемента *динамического* массива, то программа запишет единицы лишь в те 2 элемента массива, под которые была выделена память, а остальные операторы присваивания проигнорирует. А при печати элементов A[0]..A[5], то вы увидите две единицы, а за ними – мусор.

Если вы выйдете за границы массива, который расположен в *статической* памяти, то это вызовет ошибку EAccessViolation (как обрабатывать ошибки, мы изучим

позже). В TP, как вы наверное уже успели заметить, выход за границы массива никем не контролировался.

Строки в Delphi

В TP мы рассматривали с вами 1 тип строк – тип string.

В TP String был строкой фиксированной длины, в 0-м байте хранилось действительное количество символов в строке (а отличие от размера переменной, т.е. количестве байт, выделенных под размещение переменной в памяти, которые можно узнать с помощью функции sizeof).

В Delphi ситуация иная:

- Переменные типа string[n], где $n \le 255$ - это все те же паскалевские статические строки. В Delphi 2005 введен также тип ShortString, эквивалентный string[255].
- Переменные типа string – это на самом деле – ссылка на char. Конец строки – это специальный стоп-символ. Индекс 1-го элемента =0.

Кроме того, в Delphi есть специальный тип PChar, который также является указателем на char. Фактически, string и PChar – это одно и то же. При этом эти 2 типа автоматически не приводятся друг к другу. Это создает большие неудобства, которые вызваны, возможно, желанием добиться большей совместимости с TP, в котором string и PChar – разные типы данных.

Пример 3: Использование строк.

```
1 : program StringPChar;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : var
9 :   s:string;
10:   z:PChar;
11: begin
12:   s:='Andres';
13:   z:=PChar(s);
14:   writeln(z);   {Andres}
15:
16:   z[6]:='2';
17:   writeln(length(z)); { 7 }
18:   writeln(z);   {Andres2}
19:
20:   z:='Manfred';
21:   writeln(s);   {Andres}
22:   writeln(z);   {Manfred}
23:   readln;
24: end.
```

Итак, в переменных s, z хранятся ссылки на char. Т.к. s, z – ссылки, то их разыменовывать нельзя и более того, они разыменовываются автоматически, когда в них записываются значения. Например, после строки 12 переменная s будет ссылаться на строку ‘Anders’.

Может показаться, что смысл строки 13 такой: z станет указывать на ту же область памяти, что и s. Но это не так потому, что s, z – это не указатели, а ссылки. При выполнении оператора присваивания в строке 13 в область памяти на которую указывает z будет скопировано значение области памяти, на которую ссылается s. Следовательно, в z запишется копия строки, которая находится в s.

256 символов, которые позволяет закодировать 1 байт, зачастую недостаточно, особенно, когда надо кодировать иероглифы. Поэтому есть другие символьные типы. Например, символы в формате Unicode имеют размер 2 байта. Для работы с такими типами есть специальные встроенные типы данных, например WideChar. Есть и строки из таких символов, например, Widestring. В Delphi есть процедуры перевода обычных строк в WideString.

За более подробной информацией обращайтесь к справочной системе (ключевые фразы: String Types, WideString, WideChar).

Перевод чисел в строки и обратно

В Delphi переводить числа в строки надо будет гораздо чаще, чем в TP, поэтому я приведу основные процедуры, которые мы будем использовать.

```
function StrToInt(const S: string): Integer;
```
переводит строку в целое число
```
function IntToStr(Value: Integer): string; overload;
```
переводит 32-битовое целое число в строковое представление
```
function IntToStr(Value: Int64): string; overload;
```
переводит 64-битовое целое число в строковое представление
```
function StrToFloat(const S: string): Extended;
```
переводит строку в вещественное число
```
function FloatToStr(Value: Extended): string;
```
переводит вещественное число в строку

Если какая-либо из этих функций не может быть выполнена (например, если в функцию StrToInt передать строку, в которой будет записано вещественное число), то произойдет ошибка EConvertError.

Есть еще много различных методов для перевода данных из одного представления в другое. За более подробной информацией обращайтесь к справочной системе (type conversion routines, floating point conversion routines).

Модули

В описании модулей немного изменился синтаксис:

Unit Name;

interface

интерфейсная часть
implementation
исполняемая часть
initialization
часть инициализации
finalization
завершающая часть
end.

Смысл интерфейсной и исполняемой частей тот же, что и в TP. Часть инициализации в TP писалась в конце модуля между begin и end. В ней находятся операторы, которые должны быть выполнены до начала работы основной программы. В части finalization выполняются вспомогательные операторы, которые выполняются после завершения работы основной программы.

Части initialization и finalization могут отсутствовать вместе с зарезервированными словами, которыми они начинаются.

Оператор цикла for in do

В Delphi 2005 был введен новый оператор цикла – for in do. С его помощью можно просматривать все элементы массива, строки, множества или коллекции.

Пример 4: Использование оператора for in do.

```
1 : program ForInDo;
2 : {$APPTYPE CONSOLE}
3 :
4 : const
5 :   n=4;
6 : type
7 :   Mas = array[1..n] of integer;
8 : var
9 :   M:Mas;
10:   i:integer;
11: begin
12:   for i:=1 to n do
13:     M[i]:=i*5;
14:   for i in M do
15:     writeln(i);
16:   readln;
17: end.
```

Цикл в строке 14 напечатает все элементы массива M, т.е. 5, 10, 15, 20.

15.4. Работа с динамической памятью

Управление памятью в Windows

Адресация в Windows не такая, как в DOS - никаких сегментов и параграфов в ней нет. Windows использует механизм виртуальной памяти. Это означает следующее:

несмотря на то, что размер оперативной памяти ограничен, каждому процессу дается в распоряжение адресное пространство объемом 4 Гб ($=2^{32}$ байт, следовательно 4-байтовых указателей достаточно, чтобы охватить все адресное пространство). Из этих 4 Гб около 2 Гб дается в распоряжение процессу, а остальное место используется для нужд ОС. Если весь процесс может разместиться в оперативной памяти, то ОС его полностью там размещает. В противном случае часть программы записывается в специальный файл (swap-файл). Когда часть программы, сохраненная на диске, нужна, она выгружается из этого файла назад в ОП. Естественно, перенос программ из памяти на диск и обратно требуют дополнительного процессорного времени, поэтому приложения работают медленнее. Но что делать, если памяти на всех не хватает.

Адресное пространство каждого процесса (размером 4 Гб) ОС отображает на действительные адреса ОП.

Для работы с ОП в Windows есть ряд функций, которые позволяют выполнять все основные операции, связанные с выделением памяти под указатель и освобождения памяти.

Функции управления памятью в Delphi

Процедуры New, Dispose и GetMem, FreeMem сохранили свое прежнее значение (хотя их реализация изменилась, т.к. они используют функции Win32 API, а не функции DOS), а вот функций MemAvail, MaxAvail вообще нет. Вместо них есть функция, которая возвращает запись с информацией о состоянии динамической памяти:

```
function GetHeapStatus: THeapStatus;
```

Тип THeapStatus определяется так:

```
type
  THeapStatus = record
    TotalAddrSpace: Cardinal;
    TotalUncommitted: Cardinal;
    TotalCommitted: Cardinal;
    TotalAllocated: Cardinal;
    TotalFree: Cardinal;
    FreeSmall: Cardinal;
    FreeBig: Cardinal;
    Unused: Cardinal;
    Overhead: Cardinal;
    HeapErrorCode: Cardinal;
  end;
```

- **TotalAddrSpace** - объем адресного пространства, доступный приложению (в байтах).
- **TotalUncommitted** – объем части адресного пространства, для которого не выделено место в swap-файле.
- **TotalCommitted -** объем части адресного пространства, для которого выделено место в swap-файле.

- **TotalAllocated** - количество байт, которые в текущий момент использует программа (т.е. количество байтов, которые были выделены приложению и не возвращены в кучу).
- **TotalFree** – количество свободных байтов, доступных приложению.
- **FreeSmall** - общий объем «небольших» секторов, которые использовались в программе, но потом память под которые была возвращена в кучу.
- **FreeBig** - общий объем «больших» секторов, которые использовались в программе, но потом память под которые была возвращена в кучу.
- **Unused** – количество байт, которые не использовались программой (из TotalCommitted).
- **Overhead** – память, занятая администратором кучи для управления выделенной памятью.
- **HeapErrorCode** – показывает статус кучи.

Справедливы следующие равенства:
FreeBig + FreeSmall +Unused = TotalFree
TotalUncommitted + TotalCommitted = TotalAddrSpace

Пример 5: Работа с динамической памятью.

```
1 : program kap15b03;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : procedure SchreibStatus(var HS:THeapStatus;var f:text);
6 : begin
7 : with HS do
8 :   begin
9 :   writeln(f,'HeapStatus');
10:   writeln(f,'TotalAddrSpace',TotalAddrSpace);
11:   writeln(f,'TotalUncommitted',TotalUncommitted);
12:   writeln(f,'TotalCommitted',TotalCommitted);
13:   writeln(f,'TotalAllocated',TotalAllocated);
14:   writeln(f,'TotalFree',TotalFree);
15:   writeln(f,'FreeSmall',FreeSmall);
16:   writeln(f,'FreeBig',FreeBig);
17:   writeln(f,'Unused',Unused);
18:   writeln(f,'Overhead',Overhead);
19:   writeln(f,'HeapErrorCode',HeapErrorCode);
20:   end;
21: end;
22:
23: var
24:   HS:THeapStatus;
25:   p,p1,p2,p3:pointer;
26:   Dat:text;
27: begin
28:   AssignFile(Dat,'hier.txt');
29:   rewrite(Dat);
30:   HS:=GetHeapstatus; SchreibStatus(HS,Dat); writeln;
31:
```

```
32:   GetMem(p,200);
33:   GetMem(p1,200);
34:   GetMem(p2,200);
35:   HS:=GetHeapstatus; SchreibStatus(HS,Dat); writeln;
36:
37:   FreeMem(p1,200); //освобождаем средний фрагмент
38:   HS:=GetHeapstatus; SchreibStatus(HS,Dat); writeln;
39:
40:   getMem(p1,16000);  //будет подключена еще одна страница
41:   HS:=GetHeapstatus; SchreibStatus(HS,Dat); writeln;
42:
43:   FreeMem(p1,16000); // подключенная страница уберется
44:   HS:=GetHeapstatus; SchreibStatus(HS,Dat); writeln;
45:
46:   getMem(p3,2000000); // ОС добавит приложению еще памяти
47:   HS:=GetHeapstatus; SchreibStatus(HS,Dat); writeln;
48:
49:   FreeMem(p,200);
50:   FreeMem(p2,200);
51:   FreeMem(p3,2000000);
52:   HS:=GetHeapstatus; SchreibStatus(HS,Dat); writeln;
53:
54:   CloseFile(Dat);
55: end.
```

Результаты работы программы (сначала – первый столбец, затем – второй, следом – третий):

```
HeapStatus                HeapStatus                 HeapStatus
TotalAddrSpace0           TotalAddrSpace1048576      TotalAddrSpace0
TotalUncommitted0         TotalUncommitted1015808    TotalUncommitted0
TotalCommitted0           TotalCommitted32768        TotalCommitted0
TotalAllocated0           TotalAllocated16400        TotalAllocated0
TotalFree0                TotalFree16352             TotalFree0
FreeSmall0                FreeSmall204               FreeSmall0
FreeBig0                  FreeBig0                   FreeBig0
Unused0                   Unused16148                Unused0
Overhead0                 Overhead16                 Overhead0
HeapErrorCode0            HeapErrorCode0             HeapErrorCode0

HeapStatus                HeapStatus
TotalAddrSpace1048576     TotalAddrSpace1048576
TotalUncommitted1032192   TotalUncommitted1032192
TotalCommitted16384       TotalCommitted16384
TotalAllocated600         TotalAllocated400
TotalFree15768            TotalFree15972
FreeSmall0                FreeSmall204
FreeBig0                  FreeBig15768
Unused15768               Unused0
Overhead16                Overhead12
HeapErrorCode0            HeapErrorCode0
```

```
HeapStatus                  HeapStatus
TotalAddrSpace1048576       TotalAddrSpace3080192
TotalUncommitted1032192     TotalUncommitted1048576
TotalCommitted16384         TotalCommitted2031616
TotalAllocated400           TotalAllocated2000400
TotalFree15972              TotalFree31200
FreeSmall204                FreeSmall204
FreeBig0                    FreeBig0
Unused15768                 Unused30996
Overhead12                  Overhead16
HeapErrorCode0              HeapErrorCode0
```

1. Изначально программе не выделяется динамическая память.
2. Приложению выделяется 1 Мб памяти (1048576 байт) – TotalAddrSpace, из которых выделено память на диске под 16 Кб (16384 байта) – TotalCommitted, 600 байт используется программой, 16 байт занято администратором кучи.
3. Освобождается один блок из 200 байт и одновременно освобождается 4 байта, занятые ранее администратором кучи, поэтому FreeSmall = 204.
4. Для того, чтобы выделить 16000 байт под указатель p1 дополнительно выделяется память в swap-файле (еще 16 Кб).
5. Освобождается память под p1 – появляется участок памяти FreeBig = 15768. Он не равен 16000, поскольку часть памяти, которая была выделена под p1, была на другой странице. Теперь же программа снова использует лишь одну страницу, поэтому тот кусочек в учет не берется.
6. Выделяется память под указатель p3 – 2000000 байт. Для того, чтобы удовлетворить такой запрос на память, ОС выделяет дополнительную память приложению.
7. Приложение не использует больше ДП.

15.5. Использование диалогов

В принципе, приложение может и не содержать консоль, и при этом не являться нормальным оконным приложением (т.е. приложением с формой). Чтобы не использовать консоль, достаточно удалить директиву
`{$APPTYPE CONSOLE}`

При этом вы можете наладить диалог с пользователем, используя встроенные диалоговые окна, описания которых находятся в модуле Dialogs. Но процедуры readln, writeln можно будет использовать лишь для печати в файл.

Давайте рассмотрим 2 простейших диалоговых окна, которые запускаются процедурами InputBox, ShowMessage.

Пример 6: Работа с диалоговыми окнами.

```
program OhneKonsole;

uses
  SysUtils,
  Dialogs;  // для диалоговых окон
var
  mel:string;
```

```
  n:integer;
begin
  mel:=InputBox('Диалоговое окно','Введите число','0');
  n:=StrToInt(mel);
  ShowMessage(IntToStr(n*n)); //выдаем пользователю результат
end.
```

Диалоговые окна для примера 6 показаны на рис. 17.3-17.4.

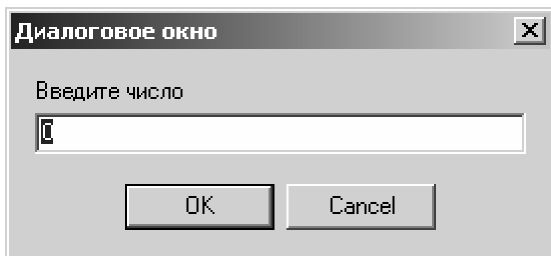


Рис 17.3. Окно ввода для примера 6

Рис 17.4. Результат, выведенный в диалоговом окне

Есть и другие встроенные диалоговые окна. Можете поискать их в справочной службе Delphi.

15.6. Работа с файлами

В Delphi изменились названия многих из процедур, которые мы использовали в TP, хотя их смысл остался прежним.

Файловые типы по прежнему делятся на бестиповые, типизированные и текстовые.

Но текстовый тип теперь называется TextFile.

Assign стала называться AssignFile.

Close стала называться CloseFile.

Процедуры writeln, readln, write, read работают так же, как и в TP.

Сейчас мы рассмотрим несколько других полезных процедур, предназначенных для работы с файлами.

```
procedure ChDir (const S: string); overload;
procedure ChDir(P: PChar); overload;
```

Устанавливает текущим каталогом тот, который указан в строке S (или P).

```
procedure GetDir(D: Byte; var S: string);
```

Записывает в переменную S полный адрес текущего каталога (D – номер диска: 0 – по умолчанию, 1- диск A, 2 – B, 3 – диск C).

```
function FindFirst(const Path: string; Attr: Integer; var F:
TSearchRec): Integer;
```

Записывает в переменную F данные о первом из файлов, которые располагаются по маршруту Path, с атрибутами Attr

```
function FindNext(var F: TSearchRec): Integer;
```

F – переменная, которая содержит информацию о файле. FindNext записывает в F информацию о следующем файле с теми же аттрибутами, что и F.

Остальные функции вы можете найти в справочной системе, написав фразу

file management routines

TSearchRec определяется так:

```
type
TSearchRec = record
     Time: Integer;
     Size: Integer;  //размер файла в байтах
     Attr: Integer;  //атрибуты файла
     Name: TFileName;  //название файла с расширением, но без пути
доступа
     ExcludeAttr: Integer;
     FindHandle: THandle;
     FindData: TWin32FindData; // вспомогательная инф.: время
создания, полное имя и т.д.
end;
```

В следующем примере мы напишем программу, которая будет искать все файлы с заданным расширением в подкаталогах.

Пример 7: Поиск файлов в директории.

```
1 : program Kap15b04;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils,
7 :   Dialogs;
8 :
9 : const
10:   n=10000;
11: type
12:   MeinTSearchRec = record
13:     TSR:TSearchRec;
14:     VollName:string; {имя файла вместе с полным путем к нему}
15:   end;
16:   DateienInfoMas = array [1..n] of MeinTSearchRec;
17:
18: {Заносит в массив M все файлы в директории Dir (включая все
19: поддиректории) с расширением Erweiterung. В TotQuant записывает
20: общее количество таких файлов.}
21: procedure AlleDateienZuMas(var M:DateienInfoMas;var
TotQuant:integer;var Dir,Erweiterung:string);
22: var
23:   sr:TSearchRec;
24:   s:string;
25: begin
26:   {$I-} //ошибки ввода-вывода надо проверять самому с помощью
IOResult
27:   ChDir(Dir); //устанавливаем новый текущий каталог
28:
29:   if IOResult<>0 then //Если каталог Dir не существует
```

```
30:     begin
31:     TotQuant:=0;
32:     exit;
33:     end;
34:   // Ищем pas-файлы в текущей директории
35:   if FindFirst('*.'+Erweiterung,faAnyFile,sr)=0 then
36:     repeat
37:       inc(TotQuant);
38:       M[TotQuant].TSR:=sr;
39:       M[TotQuant].VollName:=Dir+'\'+sr.Name;
40:     until FindNext(sr)<>0;
41:   FindClose(sr);    //освобождаем память, выд. под FindFirst
42:
43:   //Ищем подкаталоги
44:   if FindFirst('*.*',faDirectory,sr)=0 then
45:     repeat
46:       if (sr.Name<>'.') and (sr.Name<>'..') and
(sr.Attr=faDirectory)then //Если подкаталог
47:         begin
48:           s:=Dir+'\'+sr.Name;//Записываем в s путь к подкаталогу
49:           alleDateienZuMas(M,TotQuant,s,Erweiterung);
//пробегаем файлы подкаталога
50:           ChDir(Dir); // снова возвращаемся к исходному каталогу
51:         end;
52:     until FindNext(sr)<>0;
53:   FindClose(sr);  //освобождаем память, выд. под FindFirst
54: end;
55:
56: {Выдает общий объем всех файлов, содержащихся в M}
57: function NehmTotUmfang(var
M:DateienInfoMas;TotQuant:integer):integer;
58: var
59:   i,s:integer;
60: begin
61:   s:=0;
62:   for i:=1 to ToTQuant do
63:     s:=s+M[i].TSR.Size;
64:   result:=s;
65: end;
66:
67: {Печатаем все M[i] в поле для вывода}
68: procedure SchrDateien(var M:DateienInfoMas;TotQuant:integer);
69: var
70:   i:integer;
71: begin
72:   for i:=1 to TotQuant do
73:     writeln(M[i].VollName);
74:   writeln('Quantitat = ',TotQuant);
75:   writeln('Umfang (in Bytes) =',NehmTotUmfang(M,TotQuant));
76: end;
77:
78: var
```

```
79:   DatMas:dateienInfoMas;
80:   TotQuant:integer;{Количество записей в массиве}
81:   Weg,Erw:string;
82:
83: begin
84:   Weg:=InputBox('Поиск файлов в поддиректории','Введите путь к
файлам','');
85:   Erw:=InputBox('','Введите расширение (например - pas)','');
86:   AlleDateienZuMas(DatMas,TotQuant,Weg,Erw);
87:   SchrDateien(DatMas,TotQuant);
88:   readln;
89: end.
```

Разберем процедуру AlleDateienZuMas. В 26-й строке используется директива компилятора {$I-}. Она означает, что ошибки ввода-вывода, которые будут происходить при использовании файлов, придется обрабатывать самостоятельно, используя функцию IOResult. По умолчанию используется {$I+}, при которой генерируются при ошибках генерируются исключения, которые можно затем обрабатывать. Этот способ больше отвечает объектно-ориентированному подходу, но мы еще не знаем ни ООП, ни исключений, поэтому будем использовать IOResult.

IoResult возвращает 0, если ошибок не было. В программе, если указанная директория не существует, то поиск прекращается, а количество найденных файлов устанавливается равным 0. Сам алгоритм рекурсивный – чтобы найти все файлы в текущем каталоге, надо просмотреть все файлы в подкаталогах.

15.7. Файлы, связанные с проектом

Когда мы работали с TP, основная программа и программный код модулей были записані в файлах с расширением .pas, откомпилированные модули – в файлах с расширением tpu и еще, естественно, мы могли создавать .exe файл, который и был готовым приложением.

В Delphi в состав приложения входит значительно большее количество файлов:

- Основной файл проекта имеет расширение dpr (Delphi Project).
- Программный код модулей находится в файлах, имеющих расширение pas. После компиляции этих файлов создаются файлы соответствующие файлы с машинным кодом, но расширение у них не tpu, как в TP, а dcu (Delphi compiled unit).
- Если модуль содержит описание формы, то помимо pas-файла с исходным кодом создается файл с расширением dfm, в котором находится описание формы (в текстовом или двоичном виде).
- Кроме того, приложение содержит файлы ресурсов (напр. значок программы), которые присоединяются компилятором на этапе компоновки проекта к исполняемому файлу. Они имеют расширение res.
- Файлы настройки содержат параметры компилятора (с расширением cfg), проекта (с расширением bdsproj или dof в Delphi 2005 или Delphi 6,7 соответственно).

После компиляции проекта создается exe-файл, содержащий код готового приложения.

15.8. Динамически подключаемые библиотеки (DLL)

Модули, на которые разбивается программа, подключаются к основной программе статически – т.е. при компоновке программы. С помощью DLL можно подключать части программы во время ее работы. Кроме того, используя DLL, можно использовать подпрограммы, написанные на других языках программирования. Использование DLL поддерживается операционной системой Windows (для этого в Win32 API есть специальные функции). Сама Windows тоже состоит из ряда DLL.

Описание библиотек отличается от описания основной программы лишь тем, что начинается файл с исходным кодом библиотеки с ключевого слова Library, а также в библиотеке есть раздел exports, в котором перечисляются подпрограммы, которые экспортируются библиотекой (т.е. подпрограммы, которые могут использоваться в программах, которые подключают DLL).

Пример 8: Пишем простую библиотеку.

```
library MeineDLL;

function Summe(var A:array of integer):integer; stdcall;
var
  i:integer;
begin
  result:=0;
  for i:=0 to high(A) do
    result:=result+A[i];
end;

procedure ZufalligMas(var A:array of integer;k:integer); stdcall;
var
  i:integer;
begin
  for i:=0 to High(A) do
    A[i]:=random(k);
end;

procedure SchreibMas(var A:array of integer); stdcall;
var
  i:integer;
begin
  for i:=0 to High(A) do
    write(A[i],' ');
end;

Exports
  Summe index 1 name 'Summe',
  ZufalligMas index 2 name 'ZufalligMas',
  SchreibMas index 3 name 'SchreibMas';
begin
end.
```

Называется библиотека MeineDLL, следовательно, после компиляции появится файл MeineDLL.dll, который и будет подключаться к программам.

Все функции в библиотеке – вполне обычные, но объявлены с директивой stdcall для того, чтобы не возникало проблем при использовании библиотеки в программах, написанных на других языках программирования.

В конце библиотеки находится раздел exports, в котором «на экспорт» идут все 3 подпрограммы, написанные в библиотеке. После названия процедуры следует индекс, который позволяет при подключении библиотеки быстро находить адрес подпрограммы. Name – это внешнее имя подпрограммы (оно может отличаться от внутреннего, т.е. имени, которое ей дано внутри библиотеки).

15.9. Подключение библиотеки

Можно подключать функции из библиотеки просто с помощью директивы external: после полного названия подпрограммы ставится директива external и название библиотеки. Такое подключение иногда называется статическим т.к. функции после начала работы программы экспортируются из библиотеки и затем присутствуют в программе до ее завершения. Однако все равно подпрограммы подключаются не на этапе компоновки, а после запуска приложения.

Пример 9: Статическое подключение библиотеки.

```
program StatischeDLL;
{$APPTYPE CONSOLE}
const
  n=5;
var
  A:Array [1..n] of integer;

function Summe(var A:array of integer):integer; stdcall; External
'MeineDLL';
procedure ZufalligMas(var A:array of integer;k:integer); stdcall;
external 'MeineDLL';
procedure SchreibMas(var A:array of integer); stdcall;external
'MeineDLL';

begin
  randomize;
  ZufalligMas(A,45);
  SchreibMas(A);
  writeln;
  writeln(Summe(A));
  readln;
end.
```

Как видите, были подключены 3 подпрограммы из библиотеки и успешно использованы в программе.

Для того чтобы «динамически» подключать библиотеки, т.е. самим решать, когда подключать библиотеку, а когда выгружать ее из памяти, надо использовать API-

функции LoadLibrary, GetProcAddress, FreeLibrary, которые описаны в модуле Windows. Мы разберемся в том, как они работают на примере.

Пример 10: Динамическое подключение библиотеки.

```
program DynDLL;
{$APPTYPE CONSOLE}
uses
  Windows;
const
  n=5;
type
  Mas=Array [1..n] of integer;

  SummFunk = function (var A:array of integer):integer; stdcall;
  FullProz = procedure (var A:array of integer;k:integer); stdcall;
  SchrProz = procedure (var A:array of integer); stdcall;
var
  A:Mas;
  H:LongWord;
  Summe:SummFunk;
  ZufalligMas:FullProz;
  SchreibMas:SchrProz;

begin
  H:=LoadLibrary('MeineDLL.dll');
  if H=0 then
    begin
    writeln('Нет такой библиотеки');
    readln;
    exit;
    end;

  @Summe:=GetProcAddress(H,PChar(1));
  @ZufalligMas:=GetProcAddress(H,'ZufalligMas');
  @SchreibMas:=GetProcAddress(H,'SchreibMas');
  randomize;
  ZufalligMas(A,45);
  SchreibMas(A);
  writeln;
  writeln('Summe ',Summe(A));
  FreeLibrary(H);
  readln;
end.
```

В программе создаются все необходимые процедурные типы и переменные этих типов. В исполняемой части программы сначала загружается библиотека, затем в переменные функциональных типов загружаются все соответствующие функции и процедуры из библиотеки. Перед завершением работы программы, следует выгрузить библиотеку из памяти.

При динамическом подключении подпрограмм вы можете много раз загружать и выгружать различные библиотеки из памяти, что очень удобно.

15.10. Использование модулей в DLL

В DLL можно использовать модули, как и в обычной программе, но отличие в том, что внутри модуля как в разделе implementation, так и в разделе interface можно экспортировать подпрограммы (как и в самой библиотеке, с помощью exports). При создании библиотеки все модули и сама библиотека компилируются в один файл с расширением dll, и при подключении этой библиотеки из основной программы можно подключать все необходимые функции.

Инициализирующая и завершающая часть модулей, на которые ссылается библиотека будут выполнены только в случае, если на них ссылается основная программа (прямо, или косвенно, через какой-то из своих модулей).

15.11. Что нам дает DLL?

1. Не надо постоянно хранить в памяти всю программу, - можно подключать и убирать куски кода, которые уже не нужны.
2. Программа может изменять саму себя во время своей работы: часть программы, которая находится в памяти, может изменять исходный код некоторых библиотек, или создавать новые библиотеки, затем вызывать компилятор, который сможет создать из исходного кода библиотеку и затем подключать уже измененный код.
3. Привлекательна возможность написания программы на разных языках.
4. Но: библиотека может экспортировать только подпрограммы. Ни типы, ни переменные она экспортировать не в состоянии. Следовательно, классы из DLL переносить нельзя.
5. При использовании DLL следует учитывать, что если ваша подпрограмма использует модули в разных подпрограммах, то компилятор подключает к программе 1 экземпляр модуля, а если вы 2 раза подключите одну DLL в разных местах программы, то к программе будет подсоединено 2 копии одной и той же библиотеки.

Глава 16: Объектно-ориентированное программирование

16.1. Модульный подход и ООП

В модульном программировании программа состояла в общем случае из нескольких модулей, в которых сосредоточены подпрограммы, типы данных, глобальные переменные, и т.д. Разбиение программы на модули, как вы помните, - это не просто разделение одного большого файла с кодом на несколько, - модули обладают определенной самостоятельностью, т.к. у них есть внутренние подпрограммы, константы, типы …, которые невидимы в программе, которая подключает данный модуль. Но модульность принципиально не изменила подход процедурного программирования, который можно кратко выразить так: «натравить процедуру на матрицу». Локальность подпрограмм в модулях во многом напоминает локальность одной подпрограммы относительно другой.

Процедурный подход часто не соответствует нашему пониманию мира, который мы моделируем в программе: мы говорим, что машина едет, а не то, что есть функция «ехать», которая применяется к машине. Скорее у машины есть способность перемещаться, которая является ее неотъемлемой частью, иначе она – просто металлолом, а не средство передвижения. Но в таком случае мы должны каким-то образом объединить данные – саму машину (ее детали) с функциями машины – перемещаться, расходовать бензин и т.д. Именно объединение данных и функций положено в основу объектно-ориентированного программирования (ООП), которое мы подробно рассмотрим в этой главе.

Введем основные определения:

- Тип данных, в котором объединены данные и функции, называется классом.
- Экземпляр класса называется объектом этого класса.
- Подпрограммы, которые находятся внутри класса, называются методами класса.
- Данные, которые находятся в классе, называются полями.

  Позже мы рассмотрим еще одни элементы класса – свойства.
- Объектно-ориентированное программирование - методология программирования, основанная на использовании классов.
- Объединение в рамках одного типа полей и методов называется инкапсуляцией.

16.2. Как объединить данные с подпрограммами средствами процедурного программирования

Вы знаете, что программа хранится в памяти во время выполнения, следовательно, и сами подпрограммы, которые являются частью программы, тоже обладают определенным адресом (адрес подпрограммы = адрес ее начала) в оперативной памяти ПК. Кроме того, вы умеете пользоваться процедурными и функциональными типами, которые, по сути, являются ссылками на начало подпрограммы. Значит, чтобы объединить подпрограммы с данными, можно создать запись, в которой вместо каждой из функций находилась бы ссылка на некоторую подпрограмму. Безусловно, сама запись стала бы очень громоздкой, однако цели мы бы добились. Однако, для создания переменной такой записи, надо самостоятельно настроить все ссылки на подпрограммы, т.е. в каждую из них записать адрес нужной

подпрограммы. Т.к. подпрограмм-членов такой записи может быть много, то логично написать дополнительную подпрограмму-настройщицу, которая бы выполняла эту работу. Естественно, она также должна стать членом записи.

Далее будем называть запись, содержащую подпрограммы, связанные с нею, чудо-записью.

На рис. 16.1 вы видите 2 переменные типа Чудо-Запись. В каждой из них 4 поля, 2 из которых – ссылки на некоторые функции (заранее заданной сигнатуры).

Согласно нашему построению, переменные, типом которых является чудо-запись, могут менять свои функциональные свойства, т.к. чудо-запись содержит ссылки на подпрограммы, и при желании мы можем переставить их на другие подпрограммы (см. рис 16.1).

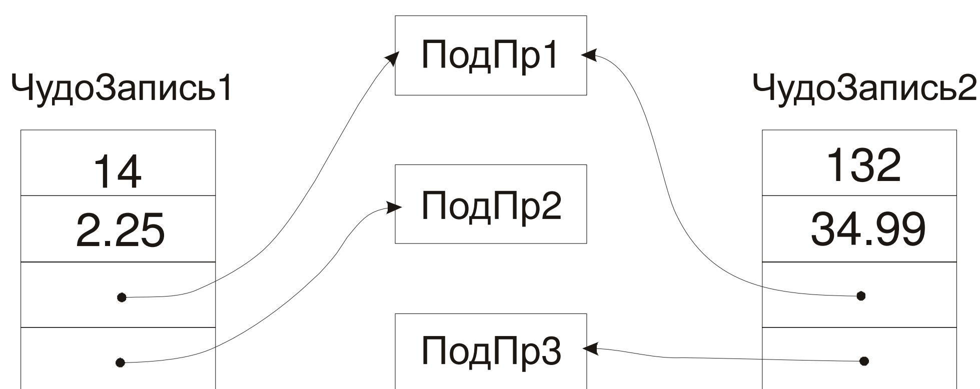


Рис 16.1 Чудо-записи, версия 1

Теперь договоримся, что набор подпрограмм у переменных, типом которых является чудо-запись, не будет изменяться на протяжении существования этих переменных. В таком случае можно будет сделать чудо-запись гораздо более компактной: для каждого типа «чудо-запись» сформировать где-то в памяти массив, который состоял бы из указателей на подпрограммы, а в самой чудо-записи будет храниться лишь указатель на этот массив. Создавать массивы для каждого типа «чудо-запись» надо в начале программы. Таким образом, чудо-записи будут отличаться по от обычных записей лишь на размер одного указателя. Заметим, что каждая подпрограмма из массива должна принимать ссылку на переменную типа «чудо-запись» (кроме тех подпрограмм, которые не используют поля чудо-записи[14]).

На рисунке 16.2. вы видите 3 переменные (ЧЗ_1, ЧЗ_2, ЧЗ_3), типом которых является некоторая чудо-запись. При этом все ссылки на подпрограммы находятся в массиве ЧЗ (все ссылки уже указывают на подпрограммы). А у элементов типа чудо-записи хранится в одном из полей указатель на этот массив.

Массив с общими для переменных типа чудо-запись подпрограммами мы назовем дескриптором чудо-записи.

Значит, в принципе реализовать принцип инкапсуляции мы можем без особых трудностей на основе двух слоев указателей (если учесть, что и подпрограммы должны принимать указатели на записи, то – трех слоев). Другие принципы ООП, которые нам еще предстоит изучить, реализовать несколько сложнее.

---

[14] В будущем мы назовем такие функции «методами класса»

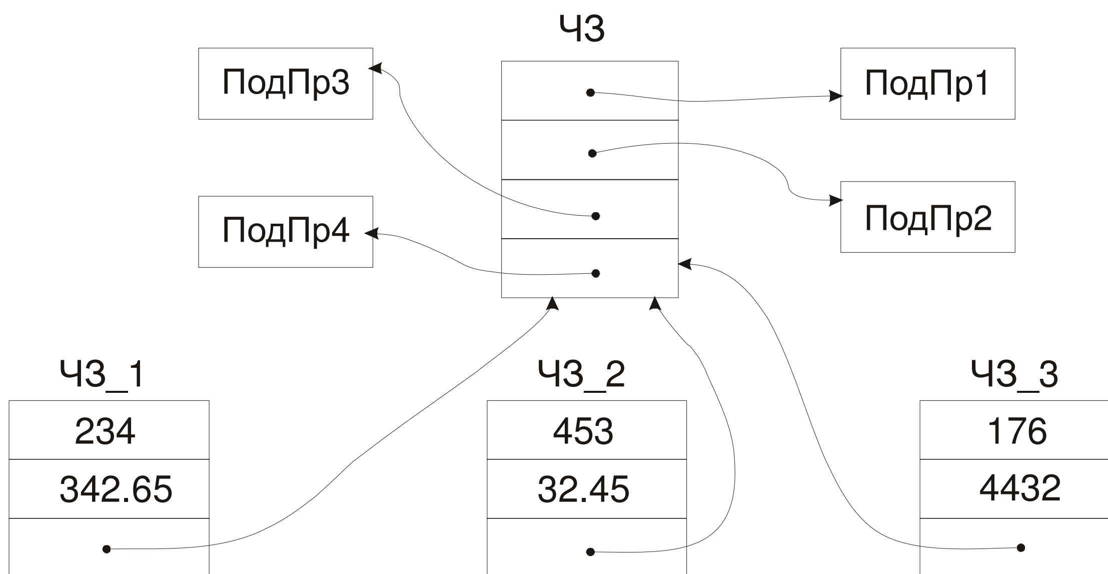


Рис 16.2 Чудо-записи, версия 2

16.3. Объявление класса. Способы доступа к элементам класса

Просто объединить данные и методы – уже хорошо, т.к. это позволяет мыслить в терминах классов и объектов. Для того чтобы сделать класс чем-то более целостным, можно ограничивать доступ к элементам класса. Это помогает обеспечить, например, непротиворечивость данных: доступ к полям будет обеспечиваться специальными функциями, которые будут проверять правильность введенных значений.

В Delphi есть 3 основных режима доступа к элементам класса – public (открытые), private (закрытые), protected (защищенные).

- private – элементы класса могут использоваться лишь внутри методов класса либо методов, которые расположены в том же модуле, где объявлено private-поле.
- public – элементы могут использоваться где угодно (так же, как и поля записи).

Режим доступа protected мы обсудим позже, когда изучим наследование.

Метод класса стоит сделать закрытым, если он является лишь подметодом других методов класса и отдельно от них использоваться не должен.

Способ доступа контролируется компилятором, поэтому если вы попытаетесь присвоить значения закрытым полям какого-то объекта вне области видимости этого поля, то компилятор выдаст сообщение об ошибке.

Объявляется класс в Delphi следующим образом:

```
type
    KlassesName = class
    private
    …  {описания private-полей и private-методов}
    public
    …  {описания public-полей и public-методов}
    end;
```

В объявлении:

KlassesName – название класса, который мы хотим создать;

class – зарезервированное слово, говорящее о том, что мы будем объявлять класс.

16.4. Конструкторы и деструкторы

Для того, чтобы создавать объекты, в каждом классе есть специальный метод, который называется конструктором (когда мы говорили о чудо-записях, то мы называли такую подпрограмму настройщицей). Любой конструктор выделяет в куче память под объект и возвращает ссылку на этот объект. Кроме этого, внутри конструктора можно выполнять и другие действия, например, заполнять поля объекта базовыми значениями.

Заметьте: создается объект в динамической памяти, а возвращается ссылка на объект. Благодаря этому вы можете работать с переменными, расположенными в динамической памяти без непосредственного использования указателей.

Объявляется конструктор следующим образом:

```
constructor Name(<параметры>);
```

<параметры> - некоторый набор параметров.

constructor – зарезервированное слово.

Name – имя конструктора.

Тип выходного параметра не описывается, т.к. и без того известно, что это ссылка на экземпляр класса.

Для того чтобы уничтожать объект из динамической памяти, у класса есть методы, называемые деструкторами.

Описывается деструктор так:

```
destructor DestrName(<параметры>);
```

Даже если вы объявите пустой класс:

```
type
  Klasse = class
  end;
```

то все равно объекты этого класса будут по умолчанию иметь стандартный набор методов. Они появляются в любом классе т.к. в Delphi существует стандартный класс TObject, у которого есть набор методов, которые должны быть, по мнению разработчиков Delphi, у каждого класса. А переносятся все эти свойства из этого класса в другие при помощи специального механизма, называемого наследованием, который мы изучим немного позже.

Пока для нас важно, что среди стандартных методов есть уже базовый конструктор Create и базовый деструктор Destroy. Create просто выделяет в куче память под объект и возвращает ссылку на него (никакой начальной инициализации нет). Destroy очищает память, занимаемую непосредственно объектом и делает ссылку равной nil. Если же объект содержит ссылки на другие объекты, которые находятся в динамической памяти, то при использовании базового деструктора Destroy эти данные будут безвозвратно утеряны. Но если вам не подходит стандартный деструктор, вы можете написать собственный.

Теперь мы напишем простейшую программу с использованием классов. В ней мы опишем простой класс Klasse, у которого есть только одно поле целого типа. Мы хотим, чтобы переменные типа Klasse не могли быть отрицательными. Следовательно, в процедуре, в которой устанавливается значение поля (StellZahl) если входной параметр отрицателен, то значение поля x устанавливается равным 0.

Пример 1: Пишем простейший класс.

```
/////////// модуль KLasseU.pas ////////////
1 : unit KlasseU;
2 :
3 : interface
4 :
5 : type
6 :   Klasse=class
7 :   private
8 :     x:integer;
9 :   public
10:     constructor Bilde(x1:integer); {строит объект}
11:     procedure StellZahl(x1:integer); {stellen - ставить}
12:     function GibZahl:integer; {GibZahl - "дай число"}
13:   end;
14:
15:
16: implementation
17:
18: constructor Klasse.Bilde(x1:integer);{строит объект}
19: begin
20:   StellZahl(x1);
21: end;
22:
23: procedure Klasse.StellZahl(x1:integer); {stellen - ставить}
24: begin
25:   if x1>=0 then
26:     x:=x1
27:   else
28:     x:=0;
29: end;
30:
31: function Klasse.GibZahl:integer; {GibZahl - "дай число"}
32: begin
33:   result:=x;
34: end;
35:
36: end.
```

Файл проекта:

```
1 : program Erste;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils,
7 :   KlasseU;
8 :
9 : var
10:   X,Y:klasse;
11: begin
12:   X:=Klasse.Bilde(-34);
```

```
13:   Y:=X;
14:   writeln(Y.GibZahl); {0}
15:   Y.StellZahl(56);
16:   writeln(Y.GibZahl); {56}
17:   writeln(X.GibZahl); {56}
18:
19:   readln;
20: end.
```

Заметьте, что при описании методов класса в части implementation перед именем метода должно ставиться имя класса, которому этот метод принадлежит и точка.

Вы видите, что параметр x объявлен в секции private, значит, он недоступен вне модуля, в котором объявлен класс. Для того чтобы устанавливать значение поля x и получать его, в классе объявлены 2 подпрограммы: StellZahl и GibZahl. В процедуре StellZahl которой сначала проверяется, будет ли число неотрицательным. Если да, то оно и записывается в поле x в качестве нового значения. В противном случае в x записывается 0.

Конструктор размещает объект в динамической памяти и возвращает указатель на объект, даже если он пустой. Однако мы хотим, чтобы при создании объекта можно было бы сразу установить его поле, поэтому и используем процедуру StellZahl.

В основной программе может возникнуть вопрос лишь о смысле 12 строки. Для того чтобы создать объект, вызывается конструктор как метод самого класса, а не объекта. Как писать методы, которые относились бы к классу в целом, мы разберем потом. Пока же запомните, что конструктор надо вызывать именно как метод класса. Таким образом, после выполнения строки 12 в X будет записана ссылка на объект класса Klasse.

После выполнения оператора присваивания в 13-й строке, Y будет указывать на тот же объект, что и X.

Операция точка в операторе Y.GibZahl означает следующее: будет разыменована ссылка Y, а затем будет вызвана процедура объекта, на который указывает Y.

16.5. Агрегация классов

Объекты, также как и записи, могут быть полями объекта другого класса. Такой способ конструирования классов называется **агрегацией**.

В следующем примере мы построим 3 класса, описывающие геометрические фигуры в трехмерном пространстве: точка, круг и шар, используя агрегацию.

Пример 2: Агрегация классов.

```
                    ///////////////// Модуль PunktU.pas ////////////////////
1 : unit PunktU;
2 :
3 : interface
4 : uses
5 :   SysUtils;//Для функции FloatToStr
6 : type
7 :   Punkt = class  //der Punkt - точка
8 :     private
9 :       x,y,z:real;
```

```
10:     public
11:       constructor Bilde(x1,y1,z1:real); //конструктор точки
12:       procedure StellPunkt(x1,y1,z1:real);//устанавливает
координаты точки
13:       function ZurZeile:string; //возвращает точку в виде строки
14:   end;
15:
16: implementation
17:
18: constructor Punkt.Bilde(x1,y1,z1:real);
19: begin
20: //Память уже выделена и указатели на подпрограммы правильно
21: // настроены. Значит, можно устанавливать начальные значения
22:   StellPunkt(x1,y1,z1);
23: end;
24:
25: procedure Punkt.StellPunkt(x1,y1,z1:real);
26: begin
27:   x:=x1;
28:   y:=y1;
29:   z:=z1;
30: end;
31:
32: function Punkt.ZurZeile:string;
33: begin
34:
ZurZeile:='('+FloatToStr(x)+';'+FloatToStr(y)+';'+FloatToStr(z)+')';
35: end;
36:
37: end.
```

////////////////// Модуль KreisU.pas /////////////////////

```
1 : unit KreisU;
2 :
3 : interface
4 :
5 : uses
6 :   PunktU,SysUtils;
7 : type
8 :   Kreis = class  //der Kreis - круг
9 :   private
10:     Zentrum:Punkt;
11:     rad:real;
12:   public
13:     constructor Bilde(x,y,z,r:real);
14:     procedure StellKreis(x,y,z,r:real);
15:     procedure StellRad(r:real);
16:     function ZurZeile:string;
17:     function GibRad:real;
18:   end;
19:
20: implementation
21: procedure Kreis.StellRad(r:real);
```

```
22: begin
23:   if r>=0 then
24:     rad:=r
25:   else
26:     rad:=0;
27: end;
28:
29: constructor Kreis.Bilde(x,y,z,r:real);
30: begin
31: //Память выделена для функций и полей, кроме вложенных объектов
32: //В данный момент Zentrum = nil, значит надо создать объект.
33:   Zentrum:=Punkt.Bilde(x,y,z);
34: //Записать вместо предыдущей строки Zentrum.StellPunkt(x,y,z)
нельзя, т.к.
35: //объект еще не существует в памяти, и функции не работают как
надо.
36:   StellRad(r);
37: end;
38:
39: procedure Kreis.StellKreis(x,y,z,r:real);
40: begin
41:   Zentrum.StellPunkt(x,y,z);
42:   StellRad(r);
43: end;
44:
45: function Kreis.ZurZeile:string;
46: begin
47:   result:=Zentrum.ZurZeile +' , Rad = '+FloatToStr(rad);
48: end;
49:
50: function Kreis.GibRad:real;
51: begin
52:   result:=rad;
53: end;
54:
55: end.
```

////////////////// Модуль KugelU.pas //////////////////////

```
1 : unit KugelU;
2 :
3 : interface
4 : uses
5 :   KreisU;
6 : type
7 :   Kugel = class //die Kugel - шар
8 :   private
9 :     HauptKreis:Kreis;//сечение шара плоскостью, проходящей через
центр шара.
10:   public
11:     constructor Bilde(x,y,z,r:real);
12:     procedure StellKugel(x,y,z,r:real);
13:     function Umfang:real;  //der Umfang - объем.
```

```
14:     function ZurZeile:string;
15:   end;
16:
17: implementation
18: constructor Kugel.Bilde(x,y,z,r:real);
19: begin
20:   HauptKreis:=Kreis.Bilde(x,y,z,r);
21: end;
22:
23: procedure Kugel.StellKugel(x,y,z,r:real);
24: begin
25:   HauptKreis.StellKreis(x,y,z,r);
26: end;
27:
28: function Kugel.ZurZeile:string;
29: var
30:   s:string;
31: begin
32:   result:=HauptKreis.ZurZeile;
33: end;
34:
35: function Kugel.Umfang:real;  //der Umfang - объем.
36: begin
37:   result:=(4/3)*Pi*HauptKreis.GibRad*
HauptKreis.GibRad*HauptKreis.GibRad;
38: end;
39:
40: end.
```

////////////////// Основной файл проекта //////////////////////

```
1 : program Agreg;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils,
7 :   PunktU in 'PunktU.pas',
8 :   KreisU in 'KreisU.pas',
9 :   KugelU in 'KugelU.pas';
10:
11: var
12:   P:Punkt;
13:   K:Kugel;
14: begin
15:   P:=Punkt.Bilde(1,2,3);//вызываем конструктор для Punkt
16:   writeln(P.ZurZeile);
17:   K:=Kugel.Bilde(1,2,5,5);//вызываем конструктор для Kugel
18:   writeln(K.ZurZeile);
19:   readln;
20: end.
```

В программе реализованы 3 класса: Punkt (точка), Kreis (круг), Kugel (шар), и написаны некоторые базовые подпрограммы к ним. Каждый класс реализован в своем модуле. Давайте сначала рассмотрим класс Punkt (файл PunktU.pas).

У этого класса есть 3 закрытых поля – пространственные координаты точки.

Кроме того, у класса есть конструктор Bilde (bilden – формировать, создавать) и 2 метода:

- StellPunkt, с помощью которого можно устанавливать координаты точки.
- ZurZeile, который записывает представление точки в виде строки.

Единственный тонкий момент во всех классах - реализация конструкторов. В дополнение к конструктору каждого из классов мы пишем еще один конструктор – Bilde. У класса Punkt есть 3 поля, которым сразу после создания объекта надо присвоить некоторые начальные значения. Для этого они должны передаваться в конструктор в качестве параметров.

Важно знать, что любой конструктор, который вы пишете сами, в начале все равно вызывает встроенный конструктор Create, поэтому даже если вы не напишете ни одной строки в конструкторе, все равно объект будет создан и ссылку на него конструктор вернет. Когда в строке 22 вызывается процедура StellPunkt, объект уже существует в памяти, поэтому его полям можно присваивать начальные значения.

Теперь давайте разберем реализацию класса Kreis. У этого класса 2 поля – радиус типа real и Zentrum – центр круга типа Punkt. В реализации методов нет ничего сложного, а реализация конструктора Bilde требует дополнительных пояснений. Как я уже писал, любой конструктор выделяет память под все поля. Это означает, что в начале реализации конструктора Bilde класса Kreis уже существует объект класса Kreis. По умолчанию все содержимое класса заполняется нулями (т.е. во всех вложенных ссылках будет находится nil). Для того, чтобы в Zentrum записать ссылку на объект класса Punkt, надо вызвать соответствующий конструктор (см. строку 33 файла Kreis.pas).

Реализацию класса Kugel вы сможете разобрать самостоятельно.

Для того чтобы достичь инкапсуляции, надо писать много функций, которые бы позволяли получать значения некоторых полей, и устанавливать их. Поэтому объем программ несколько увеличится. Но зато программа получается более структурированной и не надо волноваться, что объект может содержать недопустимые значения. При написании больших проектов лучшая структурированность важнее, чем небольшое увеличение объема исходного кода, а хорошие компиляторы позволяют свести потери при большом количестве дополнительных вызовов функций к минимуму, поэтому скорость работы приложений тоже не сильно страдает.

16.6. Наследование

Часто случается, что один класс очень похож на другой, только добавилось несколько новых полей и методов. Было бы просто замечательно перенести в новый класс каким-то образом все повторяющиеся функции. Это можно сделать с помощью наследования – механизма, аналога которого не было структурном программировании. Наследование не просто экономит время и силы программистов, но и позволяет ввести определенное родство между классами.

Для того чтобы указать, что Klasse2 порождается классом Klasse1, надо записать так:

```
type
  Klasse2 =  class (Klasse1)
  … {дополнительные элементы класса}
  end
```

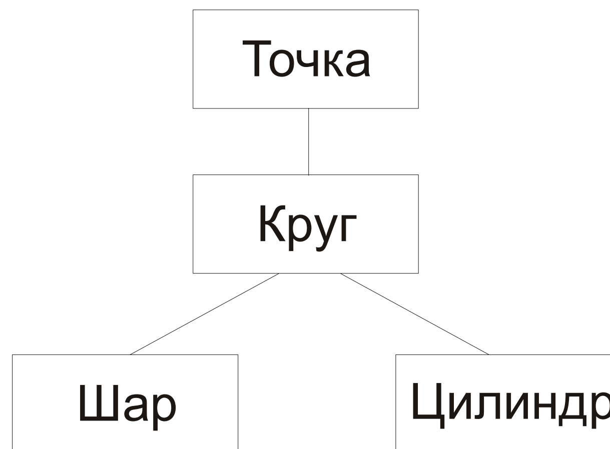


Рис 16.3: Иерархия классов геометрических фигур

При этом в Klasse2 переходят из класса Klasse1 все его методы и поля, которые являются закрытыми.

Класс, который порождается некоторым классом, называется **дочерним** классом, а тот, который порождает, - **родительским** классом. В итоге классы образуют древовидную иерархию (см. рис 16.3).

Для того чтобы элемент класса был виден не только внутри класса, в котором он описан, и классов, которые находятся в одном модуле с ним, но также во всех наследниках этого класса, создан специальный тип доступа к элементу – **protected**.

Следовательно, в класс-наследник перейдут все элементы класса родителя, которые объявлены как public или protected.

Сейчас мы построим классы Punkt, Kreis и Kugel с помощью наследования. В целях экономии места код класса Kugel писать не будем.

Пример 3: Реализация классов Punkt, Kreis, Kugel с помощью наследования

//////////////////Модуль PunktU.pas ///////////////////////////

```
1 : unit PunktU;
2 :
3 : interface
4 : uses
5 :   SysUtils;//Для функции FloatToStr
6 : type
7 :   Punkt = class  //der Punkt - точка
8 :     private
9 :       x,y,z:real;
10:     public
11:       constructor Bilde(x1,y1,z1:real); //конструктор точки
12:       procedure StellPunkt(x1,y1,z1:real);
```

```
13:       function ZurZeile:string;
14:   end;
15:
16: implementation
17:
18: constructor Punkt.Bilde(x1,y1,z1:real);
19: begin
20:   StellPunkt(x1,y1,z1);
21: end;
22:
23: procedure Punkt.StellPunkt(x1,y1,z1:real);
24: begin
25:   x:=x1;
26:   y:=y1;
27:   z:=z1;
28: end;
29:
30: function Punkt.ZurZeile:string;
31: begin
32:
ZurZeile:='('+FloatToStr(x)+';'+FloatToStr(y)+';'+FloatToStr(z)+')';
33: end;
34:
35: end.
```

///////////////////Модуль KreisU.pas ///////////////////////////

```
1 : unit KreisU;
2 :
3 : interface
4 :
5 : uses
6 :   PunktU,SysUtils;
7 : type
8 :   Kreis = class (Punkt)  //der Kreis - круг
9 :   protected
10:     rad:real; //Поле rad будет доступно в классе Kugel
11:   public
12:     constructor Bilde(x,y,z,r:real);
13:     procedure StellFigur(x,y,z,r:real);
14:     procedure StellRad(r:real);
15:     function ZurZeile:string;
16:     function GibRad:real;
17:   end;
18:
19: implementation
20: procedure Kreis.StellRad(r:real);
21: begin
22:   if r>=0 then
23:     rad:=r
24:   else
25:     rad:=0;
26: end;
```

```
27:
28: constructor Kreis.Bilde(x,y,z,r:real);
29: begin
30:  inherited Bilde(x,y,z); //Вызываем унаследованный конструктор
31:  StellRad(r);
32: end;
33:
34: procedure Kreis.StellFigur(x,y,z,r:real);
35: begin
36:   StellPunkt(x,y,z);
37:   StellRad(r);
38: end;
39:
40: function Kreis.ZurZeile:string;
41: begin
42:   result:=(inherited ZurZeile) +' , Rad = '+FloatToStr(rad);
43: end;
44:
45: function Kreis.GibRad:real;
46: begin
47:   result:=rad;
48: end;
49:
50: end.
```

/////////////////Файл проекта - ErbschaftStatisch.dpr //////////////////

```
1 : program ErbschaftStatisch;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils,
7 :   PunktU in 'PunktU.pas',
8 :   KreisU in 'KreisU.pas',
9 :   KugelU in 'KugelU.pas';
10:
11: var
12:   P:Punkt;
13: begin
14:   P:=Punkt.Bilde(1,2,3);//вызываем конструктор для Punkt
15:   writeln(P.ZurZeile); //вызовет ZurZeile класса Punkt
16:   P.Free;
17:   P:=Kreis.Bilde(1,2,4,7);
18:   writeln(P.ZurZeile); //вызовет ZurZeile класса Punkt
19:   writeln(Kreis(P).ZurZeile); //вызовет ZurZeile класса Kreis
20:   P.Free;
21:   P:=Kugel.Bilde(1,2,5,5);//вызываем конструктор для Kugel
22:   writeln(P.ZurZeile); //вызовет ZurZeile класса Punkt
23:   writeln(Kugel(P).ZurZeile); //вызовет ZurZeile класса Kugel
24:   readln;
25: end.
```

Класс PunktU.pas не изменился.

Класс Kreis – наследник класса Punkt (см. строку 8 в KreisU.pas). Однако поля x, y, z – недоступны, т.к. они объявлены как private. Как можно сделать их доступными? Два наиболее логичных способа таковы:

1. Объявить их в классе Punkt как protected, тогда x, y, z будут доступны в любом наследнике класса Punkt.
2. Написать в классе Punkt функции, которые позволили бы получать значения этих полей и сделать их public, тогда значения полей можно будет получать из любого места программы, в частности, из классов-наследников.

Теперь разберемся, какие из методов класса Punkt перейдут в класс Kreis. Private-методы, естественно, видны не будут.

При наследовании public- и protected-методов возможны 3 случая:

1. Для метода класса-родителя не существует одноименного метода в классе-наследнике. В таком случае этот метод перейдет классу-наследнику.
2. Для метода класса-родителя есть одноименный метод в классе-наследнике, но у него другая сигнатура (т.е. другой список параметров или тип возвращаемого значения). В таком случае если метод в классе-наследнике помечен директивой overload, то в классе-наследнике будут видны они оба, и будут перегруженными методами. Если директивы overload нет, то метод класса-родителя виден не будет.
3. Для метода класса-родителя есть одноименный метод в классе-потомке с точно такой же сигнатурой. В таком случае метод класса-наследника будет закрывать метод класса-родителя. При этом вы можете поставить директиву overload, и компилятор не выдаст ошибки, однако все равно метод класса-родителя виден не будет.

Согласно этим правилам в класс Kreis перейдет метод StellPunkt (это – 1-й случай). Конструктор Bilde, относящийся к классу Punkt не будет виден в классе Kreis, т.к. конструктор Bilde класса Kreis не помечен директивой overload (если бы эта директива стояла, то были бы видны оба одноименных конструктора, т.к. у них разная сигнатура). Но, несмотря на то, что конструктор Bilde класса Punkt не виден внутри класса Kreis, получить доступ к нему внутри класса Kreis можно, используя специальное ключевое слово **inherited**. Так, в строке 30 вызывается конструктор Bilde класса-родителя, а затем устанавливается свойство rad.

А заголовок функции ZurZeile, объявленной в классе Punkt, полностью совпадает с заголовком функции ZurZeile, заданной в классе Kreis. Следовательно, в классе Kreis будет видна лишь ZurZeile, объявленная в нем самом.

Давайте разбираться с файлом проекта.

В строке 12 объявляется P – ссылка на объект класса Punkt (т.е. ссылка на содержимое класса Punkt). В строках 14-15 ничего удивительного нет. В 18-й строке объект, на который ссылается P, удаляется из памяти с помощью функции Free (если ссылка не равна nil, то эта функция вызывает деструктор объекта и затем устанавливает значение ссылки равным nil, в противном случае метод Free ничего не делает).

В 17-й строке создается объект класса Kreis и ссылка на него записывается в P. Такое присваивание вполне допустимо, т.к. Kreis – наследник Punkt (подробнее о совместимости объектов будет рассказано позже). Но, несмотря на то, что теперь P

ссылается на содержимое объекта класса Kreis, компилятор будет рассматривать P как ссылку на объект класса Punkt, и в строке 18 все равно будет вызвана функция ZurZeile класса Punkt.

Для того, чтобы вызвать ZurZeile для класса Kreis, мы должны явно привести P к типу Kreis (т.е. рассматривать P как ссылку на объект класса Kreis), что и сделано в строке 19. Дальше все должно быть понятно без комментариев.

16.7. Виртуальные функции, полиморфизм и динамическое связывание

Внутри классов, которые образуют иерархию, методы могут переопределяться в классах-наследниках. Но если ссылке на базовый класс мы присвоим ссылку на класс-наследник то, как вы видели из предыдущего примера, вызываться будет всегда функция того класса, который указан для данной ссылки в разделе var. Это зачастую неудобно: было бы очень заманчиво сделать так, чтобы при вызове методов, общих для всех наследников базового класса, вызывались методы именно того класса, на объект которого указывает ссылка, а не методы класса, который указан при объявлении ссылки. Такой метод переопределения методов называется динамическим связыванием методов.

Введем важное определение:

- Полиморфизм – способность одного и того же выражения принимать различные значения (в зависимости от некоторых других факторов).

В ООП полиморфизм проявляется в том, что метод, который вызывается при помощи ссылки, зависит от класса объекта, на который указывает ссылка.

Сейчас мы реализуем те же классы Punkt, Kreis и Kugel, но с помощью динамического связывания.

Пример 4: Использование динамического связывания. Части implementation во всех модулях точно такие же, как и в примере №3, поэтому приводятся только части interface модулей и файл проекта.

```
1 : unit PunktU;
2 :
3 : interface
4 : uses
5 :   SysUtils;//Для функции FloatToStr
6 : type
7 :   Punkt = class  //der Punkt - точка
8 :     private
9 :       x,y,z:real;
10:     public
11:       constructor Bilde(x1,y1,z1:real); //конструктор точки
12:       procedure StellPunkt(x1,y1,z1:real);
13:       function ZurZeile:string; virtual;  //Виртуальная функция
14:   end;
```

//////////////////Модуль KreisU.pas ///////////////////////////

```
1 : unit KreisU;
2 :
3 : interface
4 :
```

```
5 : uses
6 :   PunktU,SysUtils;
7 : type
8 :   Kreis = class (Punkt)  //der Kreis - круг
9 :   protected
10:     rad:real; //Поле rad будет доступно в классе Kugel
11:   public
12:     constructor Bilde(x,y,z,r:real);
13:     procedure StellFigur(x,y,z,r:real);
14:     procedure StellRad(r:real);
15:     function ZurZeile:string; override;  //перекрываем метод
16:     function GibRad:real;
17:   end;
```

//////////////////Модуль KugelU.pas ///////////////////////////

```
1 : unit KugelU;
2 :
3 : interface
4 : uses
5 :   KreisU;
6 : type
7 :   Kugel = class (Kreis) // шар - наследник круга
8 :   public
9 :     constructor Bilde(x,y,z,r:real);
10:     function Umfang:real;  //der Umfang - объем.
11: // Функции печати и установки свойств переходят в Kugel без
изменений
12:   end;
25: end.
```

////////////////// Файл проекта ///////////////////////////

```
1 : program Erbschaft;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils,
7 :   PunktU in 'PunktU.pas',
8 :   KreisU in 'KreisU.pas',
9 :   KugelU in 'KugelU.pas';
10: type
11:   PunktMas = array[1..3] of Punkt;
12: var
13:   PM:PunktMas;
14:   i:integer;
15: begin
16:   PM[1]:=Punkt.Bilde(1,2,3);
17:   PM[2]:=Kreis.Bilde(1,2,4,7);
18:   PM[3]:=Kugel.Bilde(1,2,5,5);
19:   for i:=1 to 3 do
20:     writeln('PM[',i,'] = ',PM[i].ZurZeile);
```

```
21:   readln;
22: end.
```

В описании класса Punkt изменилась по сравнению с предыдущим примеров только строка 13: после заголовка метода ZurZeile стоит директива virtual. Это означает, что функция объявлена как виртуальная функция. Если функция виртуальна, то ее можно переопределять в классах-наследниках. Для этого после переопределяемой функции надо поставить директиву override (смотрите KreisU.pas строка 15). В классе Kugel функция не переопределяется, поэтому она переходит в неизменном виде от класса Kreis, наследником которого является Punkt.

В файле проекта создается массив из трех ссылок на класс Punkt. Но первая ссылка будет указывать на объект класса Punkt, вторая – на объект класса Kreis, а третья – на объект класса Kugel.

Запустив программу, вы увидите, что в строках 19-20 при вызове функции ZurZeile вызывались методы именно тех классов, на объекты которых указывают ссылки.

16.8. Смешанные иерархии

Под смешанной иерархией классов мы будем понимать иерархию классов, в которых один и тот же метод может перекрываться иногда статически, а иногда динамически. В примере мы рассмотрим тонкости применения смешанных иерархий.

Пример 5: Смешанные иерархии классов.

```
1 : program VirtUndStat;
2 : {$APPTYPE CONSOLE}
3 :
4 : type
5 :   KL = class
6 :     procedure MacheEtwas;virtual;
7 :   end;
8 :
9 :   KLKind = class(KL)     //Kind - ребенок
10:     procedure MacheEtwas;override;
11:   end;
12:
13:   KLEnkel = class(KLKind) // Enkel - внук
14:     procedure MacheEtwas;virtual;
15:   end;
16:
17:   KLGrossEnkel = class(KLEnkel)  //GrossEnkel - правнук
18:     procedure MacheEtwas;override;
19:   end;
20:
21: procedure KL.MacheEtwas;
22: begin
23:   writeln('KL')
24: end;
25:
26: procedure KLKind.MacheEtwas;
```

```
27: begin
28:   writeln('KLKind')
29: end;
30:
31: procedure KLEnkel.MacheEtwas;
32: begin
33:   writeln('KLEnkel')
34: end;
35:
36: procedure KLGrossEnkel.MacheEtwas;
37: begin
38:   writeln('KLGrossEnkel')
39: end;
40:
41: var
42:   F:KL;
43:   F2:KLEnkel;
44: begin
45:   F:=KL.Create;            F.MacheEtwas; F.Free;//KL
46:   F:=KLKind.Create;        F.MacheEtwas; F.Free;//KLKind
47:   F:=KLEnkel.Create;       F.MacheEtwas; F.Free;//KLKind
48:   F:=KLGrossEnkel.Create;  F.MacheEtwas; F.Free;//KLKind
49:
50:   F2:=KLEnkel.Create;     F2.MacheEtwas;F2.Free;//KLEnkel
51:   F2:=KLGrossEnkel.Create;F2.MacheEtwas;F2.Free;//KLGrossEnkel
52:
53:   readln;
54: end.
```

Итак, в базовом классе KL объявляется виртуальный метод MacheEtwas, который перекрывается динамически в его непосредственном потомке KLKind. В потомке класса KLKind – KLEnkel метод MacheEtwas снова объявляется виртуальным. Это означает, что иерархия виртуальных методов на классе KLEnkel прервется, т.к. в этом классе метод перекрывается статически, а директива virtual говорит о том, что в дальнейшем классы-потомки могут перекрывать виртуальный метод MacheEtwas класса KLEnkel, что и делается в классе KLGrossEnkel.

В комментарии к строкам 45..51 приводятся значения, которые будут напечатаны функциями MacheEtwas. Обратите внимание на применение функции Free к ссылкам. Если бы, например, в строке 45 не стоял бы оператор F.Free, то после выполнения первого из операторов строки 46 (конструктора) переменная F будет указывать уже на другой объект в памяти, а предыдущий объект будет продолжать существовать в динамической памяти, засоряя ее. Чтобы этого не произошло, вызывается функция Free, которая уничтожает объект из ДП (и устанавливает значение ссылки равным nil).

16.9. Иерархия классов Delphi

Вы знаете уже, что у любого объекта есть встроенные методы (и свойства). Они появляются т.к. каждый класс Delphi является наследником класса TObject, даже если это не указывается явно.

Объявление класса

```
KL1 = class
end;
```

эквивалентно следующему:

```
KL1 = class (TObject)
end;
```

На рисунке вы можете видеть небольшой фрагмент иерархии встроенных классов Delphi:

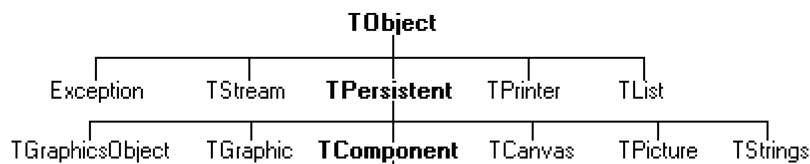


Рис 16.4: Фрагмент иерархии классов Delphi.

Методы класса TObject мы обсудим позднее. Но кроме набора методов у объекта класса TObject есть 2 поля. Что это за поля станет ясно, если вспомнить чудо-записи. Первое – это ссылка на дескриптор класса (это тот массив, на который ссылалось одно из полей чудо-записи). Второе – это ссылка объекта на самого себя. Как вы помните, функции чудо-записи (и класса) должны принимать в качестве одного из параметров ссылку на объект. Но дать возможность передавать его явно нельзя, т.к. это нарушает инкапсуляцию классов: можно вызвать функцию одного объекта, а в качестве параметра передать ссылку на другой объект. Для того, чтобы избавиться от этих проблем, у каждого объекта есть поле, хранящее ссылку на самого себя. Она автоматически передается в метод в качестве параметра.

Поле, хранящее ссылку на класс, скрыто от программиста (хотя, как мы увидим, получить ссылку на дескриптор класса можно с помощью метода ClassType класса TObject). А ссылку объекта на самого себя можно использовать внутри класса, и называется она self.

16.10. Абстрактные классы

Иногда имеет смысл создавать на вершине иерархии классы, часть методов которых не имеет реализации, но которые переопределяются в классах-наследниках.

- Виртуальные методы, которые не имеют реализации, называются **абстрактными**.
- Класс, который содержит хотя бы один абстрактный метод, называется **абстрактным**.

Пример 6: Использование абстрактных классов.

```
1 : {$APPTYPE CONSOLE}
2 :
3 : uses
4 :   SysUtils;
5 :
6 : type
7 :   Abstr = class
8 :   public
```

```
9 :     function ZurZeile:string; virtual;abstract;
10:   end;
11:
12:   Punkt = class(Abstr)
13:   private
14:     x,y:integer;
15:   public
16:     constructor Bilde(x,y:integer);
17:     function ZurZeile:string; override;
18:   end;
19:
20: function Punkt.ZurZeile:string;
21: begin
22:   result:='('+IntToStr(x)+';'+IntToStr(y)+')';
23: end;
24:
25: constructor Punkt.Bilde(x,y:integer);
26: begin
27:   self.x:=x;
28:   self.y:=y;
29: end;
30:
31: var
32:   A:Abstr;
33: begin
34:   A:=Abstr.Create;
35:   A.Free;
36: //  A.ZurZeile;  {Так нельзя - будет ошибка EAbstractError}
37:   A:=Punkt.Bilde(3,4);
38:   writeln(A.ZurZeile);
39:   readln;
40: end.
```

В строке 7 объявляется абстрактный класс, в котором всего один метод (помимо тех, которые он унаследовал от TObject). Метод ZurZeile объявлен как виртуальный и абстрактный, следовательно, его реализацию писать не надо. Кроме класса Abstr объявляется его наследник, в котором переопределяется метод ZurZeile (строка 17).

Заметьте, что можно создавать объекты абстрактного класса, работать с ними, однако ни в коем случае нельзя вызывать абстрактные методы (ведь у них реализации даже нет). Если абстрактный метод будет вызываться, то произойдет ошибка EAbstractError.

Возникает вопрос, зачем вообще разрешать создавать объекты абстрактных классов, если некоторые их методы нельзя использовать, и вводить из-за этого лишнюю ошибку EAbstractError?

Думаю, что смысл в том, что абстрактные классы обычно находятся на вершине иерархии классов. Поэтому если вы хотите создать набор объектов, среди которых могут быть объекты любого из потомков этого абстрактного класса, то логично объявить в разделе var набор ссылок на объекты абстрактного класса, которые в процессе работы программы будут хранить ссылки на объекты производных классов.

За счет полиморфизма у всех этих объектов можно было бы вызывать методы соответствующих классов.

Возможно, следовало бы запретить вызов конструктора для абстрактных классов, не запрещая объявления ссылок на объекты абстрактных классов. Тогда и проблем было бы меньше, и все преимущества бы остались.

16.11. Методы класса

У класса могут быть методы, реализация которых не зависит от полей класса. Следовательно, такие методы могут вообще не принимать в качестве параметра ссылку на объект. А так как ссылка на объект для таких методов не важна, то имеет смысл дать возможность вызывать такие методы, ссылаясь лишь на класс, к которому они принадлежат, а не на какой-то конкретный объект. В Delphi есть возможность создавать такие методы, которые называются методами класса. Для того чтобы объявить методы класса, надо перед словом function (или procedure) добавить слово class.

Конструктор – тоже метод класса, но по синтаксису перед ним слово class не ставится, т.к. он и так особенный.

Пример 7: Использование методов класса.

```
{$APPTYPE CONSOLE}

type
  MFunk = class
    class function tan(x:real):real;
  end;

class function MFunk.tan(x:real):real;
begin
  result:=sin(x)/cos(x);
end;

begin
  writeln(MFunk.tan(Pi/6));
  readln;
end.
```

В первой строке исполняемой части вызывается функция tan класса MFunk. Фактически, называние класса играет роль ссылки на дескриптор класса.

16.12. Класс TObject

Т.к. любой класс является наследником класса TObject, то надо знать основные методы этого класса.

Вы уже знаете, что у класса TObject есть конструктор Create и деструктор Destroy, которые выполняют все необходимые действия по созданию/удалению объекта любого класса, а также метод Free.

Давайте рассмотрим метод класса TObject
class function ClassName: ShortString;

Если этот метод вызывается самим классом (т.е. ссылкой на его дескриптор), то он вернет название этого класса. Если метод вызывается ссылкой на объект, то он вернет название класса, которому принадлежит объект (а не название класса, который был указан при объявлении ссылки).

Пример 8: Использование метода ClassName.

```
1 : program ClassnameFunk;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   Klasse = class
10:   end;
11:
12: var
13:   K:Klasse;
14:   Obj:TObject;
15: begin
16:   K:=Klasse.Create;
17:   Obj:=Klasse.Create;
18:   writeln(K.ClassName);      //Klasse
19:   writeln(TObject(K).ClassName); //Klasse
20:   writeln(Obj.ClassName);  //Klasse
21:   Obj.Destroy;
22:   Obj:=TObject.Create;
23:   writeln(Obj.ClassName); //TObject
24:   readln;
25: end.
```

После строк 16-17 переменные К и Obj ссылаются на объект класса Klasse. Как видно из строк 18 и 20, название класса, которому принадлежит объект, не зависит от типа, который был указан при объявлении ссылки. Приведение переменной К к другому типу (строка 19) также не влияет на значение метода ClassName.

В строках 21-23 уничтожается объект (удаляется из динамической памяти) и вместо этого создается новый экземпляр класса TObject. Как видите, теперь имя класса будет TObject (стр. 23).

Т.к. этот метод есть у любого класса, то можно в процессе работы программы сортировать объекты по классам, что очень удобно. В будущем мы узнаем еще более важные методы класса TObject, которые позволяют получить ссылки на дескриптор класса и на дескриптор класса-родителя.

16.13. Приведение классов. Операторы is, as

- Класс К1 совместим по типу с классом К2, если К1 является наследником К2 (неважно – прямым, или есть промежуточные классы). В противном случае классы К1 и К2 несовместимы по типу.

Хотя в Delphi можно приводить типы с помощью обычной операции приведения типов, однако она считается потенциально опасной, т.к. программист может попытаться привести объект к классу, с которым тип объекта несовместим, и тогда может произойти ошибка.

Для того чтобы можно было безопасно приводить классы, введен оператор **as**. Дело в том, что тип любого объекта можно определить во время работы программы, поэтому можно и определить, является ли класс, к которому приводится объект, совместимым с типом объекта. Если уже на этапе компоновки компилятор видит, что приведение к типу провести нельзя, то он выдаст сообщение об ошибке. Если же приведение типов возможно, то этот оператор его выполнит.

Кроме того, на этапе прогона программы с помощью оператора **is**[15] можно проверять принадлежность объекта некоторому классу.
A is B выдает True, если A – наследник B.

В следующем примере показан синтаксис и тонкости применения этих операторов.

Пример 9: Использование операторов is, as и обычное приведение типов.

```
1 : program IsAs;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   Kl1 = class
10:   end;
11:   Kl2 = class
12:   end;
13:
14:   var
15:   K1:Kl1;
16:   K2:Kl2;
17:   Obj:TObject;
18:   p:pointer;
19: begin
20:   K1:=Kl1.Create;
21:   K2:=Kl2.Create;
22:   Obj:=Kl2.Create;
23:
24:   if (K1 is TObject) then   //TRUE
25:     writeln('K1 ist TObject');
26:   if (Obj is Kl1) then      //FALSE
27:     writeln('Obj ist Kl1');
28:   if (Obj is Kl2) then      //TRUE
29:     writeln('Obj ist Kl2');
30:
```

[15] Вообще говоря, is – это не оператор, а операция, которая по ссылке на объект (или на дескриптор класса) и названию класса (т.е. ссылки на его дескриптор) возвращает результат булевского типа.

```
31:   if (Kl1(K2) is Kl1) then  //TRUE
32:     writeln('Kl1(K2) ist Kl1');
33:   writeln('Wirklicher Name:', KL1(K2).ClassName);//настоящее имя
34: {    //Если раскомментировать этот фрагмент, то компилятор
выдаст ошибку
35:   if ((K2 as Kl1) is Kl1) then
36:     writeln('Kl1(K2) ist Kl1');
37: }
38:   p:=Kl1.Create;
39:   if TObject(p) is Kl1 then  //TRUE
40:     writeln('Name ',TObject(p).ClassName);
41:   readln;
42: end.
```

Т.к. Obj ссылается на объект класса Kl2, то условие в строке 26 не выполнится, а условие в строке 28 будет верным.

Заметьте (строка 31), что хотя объект K2 принадлежит классу Kl2, но если его привести к типу Kl1, то условие в строке 31 выполнится. А функцию ClassName этим не обманешь – она все равно выдаст настоящее имя. Убрав комментарий в строках 34-37 и запустив программу, вы убедитесь, что компилятор выдаст ошибку на этапе компоновки программы и до выполнения дело даже не дойдет.

В строке 38 был создан объект класса Kl1 и его адрес в динамической памяти был занесен в бестиповый указатель. При этом можно без проблем приводить этот указатель к типу TObject и потом проверять принадлежность объекта, адрес которого находится в переменной p на принадлежность типу Kl1. В данном случае мы приводим бестиповый указатель к ссылке на объект класса TObject, что можно сделать лишь обычным приведением типов; оператор as в данном случае использовать нельзя, т.к. компилятор на этапе компоновки скажет вам, что бестиповый указатель приводить к классу нельзя.

Вы видите, что смешанное использование операции приведения типов и операторов is, as приводит к неожиданным результатам, поэтому гораздо лучше не использовать бестиповые указатели как ссылки на объекты – для этого объявляйте ссылку на TObject и старайтесь не использовать операцию приведения типов при работе с объектами.

16.14. Метаклассы

Часто нужна ссылка не на объект, а на дескриптор класса. Вы уже знаете, что название класса служит в коде ссылкой на дескриптор этого класса. Но в таком случае вы можете ссылаться лишь на дескриптор какого-то конкретного класса. Для того, чтобы достичь общности, в Delphi вводится понятие метакласса:

- Метакласс – тип данных, переменной которого являются ссылки на дескрипторы классов.

С помощью метаклассов можно передавать в качестве параметров методов ссылки на дескрипторы классов.

Описываются метаклассы так:

```
MetaKlName = class of Etwas
```

Где MetaKlName – это название метакласса, а Etwas – название любого класса.

Например:

```
type
  MetaKlasse = class of TControl
```

Значением объекта этого метакласса (метаобъекта) может быть ссылка на дескриптор любого класса, являющегося наследником TControl, либо самим TControl.

Пример 10: Класс, порождающий объекты других классов.

```
1 : {$APPTYPE CONSOLE}
2 : uses
3 :   SysUtils;
4 : type
5 :   MetaKlasse = class of TObject;
6 :
7 :   KLasse = class
8 :   public
9 :     x:integer;
10:   end;
11:
12:   AufBauer = class
13:   public
14:     class function GibObjekt(MK:MetaKlasse):TObject;
15:   end;
16:
17: class function AufBauer.GibObjekt(MK:MetaKlasse):TObject;
18: begin
19:   Result:=MK.Create;
20: end;
21:
22: var
23:   MK:MetaKlasse;
24:   k:TObject;
25: begin
26:   MK:=KLasse;
27:   k:=Aufbauer.GibObjekt(MK);
28:   Klasse(k).x:=10;
29:   writeln(Klasse(k).x);
30:   readln;
31: end.
```

В строке 5 объявляется метакласс, метаобъектом которого может быть ссылка на дескриптор любого класса. В строке 12 объявляется класс Aufbauer (aufbauen – сооружать, создавать, соответственно Aufbauer – конструктор, создатель). У этого класса есть только 1 метод, причем – метод класса, который принимает метаобъект (содержащий ссылку на класс) и возвращает ссылку на TObject. Все, что делает эта функция сводится к вызову конструктора класса, ссылку на который она получила в качестве параметров. Кстати, сам объект класса AufBauer может быть создан той же функцией GibObjekt. Так что с помощью метаклассов можно вообще избавиться от вызова конструкторов в коде, а вместо этого всегда поручать это дело ауфбауеру.

Ясно, что по определению нельзя объявить метакласс метакласса, кроме того, нельзя наследовать от метакласса (наследовать можно лишь от обычного класса).

Ссылку на дескриптор класса объекта, на который ссылается объектоссылка и ссылку на дескриптор родителя этого класса можно определить с помощью методов, которые определены в классе TObject:

```
type
  TClass = class of TObject;

function ClassType: TClass; //возвращает тип объекта
class function ClassParent: TClass; //возвращает тип родителя
объекта
```

Т.к. метод ClassParent объявлен как метод класса, то можно на этапе прогона программы узнавать родителя любого класса. Делать метод ClassType методом класса смысла нет, т.к. тогда получится, что вы используете ссылку на класс для получения ссылки на этот же класс.

В следующем примере вводится наследник класса TObject, который содержит дополнительный метод класса, который возвращает уровень вложенности класса (TObject считается первым уровнем).

Пример 11: Уровень вложенности класса.

```
1 : program ClassPar;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   NeuesObjekt = class(TObject)
10:     class function Niveau:integer; //возвращает уровень
вложенности класса
11:   end;
12:
13:   Klasse = class(NeuesObjekt)
14:   end;
15:
16: class function NeuesObjekt.Niveau:integer;
17: var
18:   KL:TClass;
19:   n:integer;
20: begin
21:   kl:=ClassParent;
22:   n:=1;
23:   while (kl<>nil) do
24:     begin
25:     kl:=kl.ClassParent;
26:     inc(n);
27:     end;
28:   result:=n;
29: end;
```

```
30:
31:
32: begin
33:   writeln(Klasse.Niveau);  // 3
34:   readln;
35: end.
```

16.15. Методы метаклассов и метаобъектов

Пусть объявлен следующий метакласс:

```
type
  MetaKl = class of Etwas;
```

Тогда у самого метакласса MetaKl и любого его метаобъекта методами будут все class-методы класса Etwas. Методы класса, так же, как и обычные методы, могут переопределяться статически и динамически.

Пример 12: Статическое переопределение методов метаклассов.

```
1 : program MetaStat;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : type
6 :   neu = class
7 :   public
8 :     class function Etwas:integer;
9 :   end;
10:
11:   NeuKind = class(Neu)
12:   public
13:     class function Etwas:integer;
14:   end;
15:
16:   MetaNeu = class of Neu; //объявляем метакласс
17:
18: class function Neu.Etwas:integer;
19: begin
20:   result:=10;
21: end;
22:
23: class function NeuKind.Etwas:integer;
24: begin
25:   result:=121;
26: end;
27:
28: var
29:   MN:MetaNeu;
30: begin
31:   MN:=Neu;
32:   writeln(MN.Etwas); //10
33:   MN:=NeuKind;
```

```
34:   writeln(MN.Etwas);  //10
35:   readln;
36: end.
```

Пример 13: Динамическое переопределение методов метаклассов.

```
1 : program MetaVirt;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : type
6 :   neu = class
7 :   public
8 :     class function Etwas:integer;virtual;
9 :   end;
10:
11:   NeuKind = class(Neu)
12:   public
13:     class function Etwas:integer;override;
14:   end;
15:   MetaNeu = class of Neu;
16:
17: class function Neu.Etwas:integer;
18: begin
19:   result:=10;
20: end;
21:
22: class function NeuKind.Etwas:integer;
23: begin
24:   result:=121;
25: end;
26:
27: var
28:   MN:MetaNeu;
29: begin
30:   MN:=Neu;
31:   writeln(MN.Etwas); //10
32:   MN:=NeuKind;
33:   writeln(MN.Etwas);  //121
34:   readln;
35: end.
```

16.16. Свойства

Кроме полей и методов у класса есть еще свойства. Они используются для регулирования доступа к полям класса.

Давайте разберемся в том как они работают на примере. В нем мы создадим класс MatrKarte, у которого будет 2 поля – фамилия матроса и фамилия капитана корабля. Чтобы удовлетворить требованию инкапсуляции, надо сделать эти поля закрытыми. Но получать доступ к ним можно будет с помощью свойств, а не методов.

Пример 14: Свойства.

```
1 : program EigenSch;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : type
9 :   MatrKarte = class //карточка матроса
10:   private
11:     Matr:string; //фамилия матроса
12:     Kap:string; //фамилия капитана корабля
13:
14:     procedure SchrZeile(z:string);
15:     function gibKapit:string;
16:   public
17:     constructor Bilde(const MatrName,KapitName:string);
18:     property Matrose:string read Matr write SchrZeile;
19:     property Kapitan:string read GibKapit;
20:   end;
21:
22: constructor MatrKarte.Bilde(const MatrName,KapitName:string);
23: begin
24:   matrose:=MatrName; //Но не matr:=MatrName;
25:   Kap:=KapitName;  //Но не Kapitan:=KapitName;
26: end;
27:
28: procedure MatrKarte.SchrZeile(z:string);
29: begin
30:   if z<>'AAA' then
31:     self.Matr:=z
32:   else
33:     self.Matr:='ZZZ';
34: end;
35:
36: function MatrKarte.gibKapit:string;
37: begin
38:   result:=Kap;
39: end;
40:
41: var
42:   K:MatrKarte;
43:   i:integer;
44: begin
45:
46:   K:=MatrKarte.Bilde('AAA','Schlachter');
47:   writeln(K.Matrose);
48:   K.Matrose:='AAD';
49:   writeln(K.Matrose);
50:   writeln(K.Kapitan);
51:
52:   readln;
```

```
53: end.
```

В строчках 18-19 объявляются 2 свойства: Matrose, Kapitan (правильно - Kapitän). Обращаться к свойствам можно с помощью операции точка, как к полям и методам. Свойства могут быть доступны либо для чтения, либо для записи, либо и для того и для другого. Способ доступа определяется методами или полями, которые находятся после ключевых слов read, write в описании свойства.

Например, в строке 18 описано свойство Matrose. После слова read стоит название поля Matr. Это означает, что при попытке прочитать данные свойства Matrose будет выдано значение поля Matr. А после слова write стоит название метода SchrZeile. Это означает, что при попытке присвоить свойству Matrose какое-то значение, будет автоматически вызван метод SchrZeile, параметром которого будет строка, которую вы хотите присвоить значению свойства Matrose.

Свойство Kapitan доступно только для чтения.

Кстати, если свойство доступно и для чтения, и для записи, то по синтаксису вы должны сначала описать действия при чтении, а потом – действия при записи.

Например, следующее описание было бы неверным:

```
property Matrose:string write SchrZeile read Matr;
```

16.17. Индексированные свойства

Можно объявлять и индексированные свойства. Для этого после названия свойства надо в квадратных скобках указать параметр (или параметры), согласно которым будут индексироваться элементы массива, например:

```
property Mass[Ind:integer]:integer read GibZahl write StellZahl;
property Mass[i,j:integer]:TObject read GibObj write StellObj;
```

Если свойство индексировано, то при чтении и записи свойства можно ссылаться лишь на методы, причем набор параметров должен быть таким:

Функция, отвечающая за чтение (read), должна содержать параметры, указанные в квадратных скобках (в том же порядке следования).

Процедура, с помощью которой надо устанавливать значения полей, должна содержать параметры, указанные в квадратных скобках в порядке их следования и затем еще один параметр того же типа, что и свойство.

В следующем примере мы напишем класс, который представляет собой обертку для целочисленного массива, причем этот массив может содержать лишь положительные числа. У класса будет индексированное свойство, а функции, с помощью которых можно будет получать значения элементов массива будут проверять, не вышел ли элемент за границы массива и будет ли он неотрицательным. Если пользователь запрашивает элемент массива с несуществующим индексом, то метод возвращает в качестве результата 0.

Пример 15: Использование индексированных свойств.

```
1 : program ArrProp;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
```

```
6 :   SysUtils;
7 :
8 : const
9 :   n=5;
10: type
11:   Mas = array [1..n] of integer;
12:
13:   MasKlass = class
14:   private
15:     R:Mas;
16:     procedure StellZahl(Index:integer;G:integer);
17:     function GibZahl(Index:integer):integer;
18:   public
19:     property Mass[Ind:integer]:integer read GibZahl write
StellZahl;
20:     function ZurZeile:string;  //преобразует в MasKLass строку
21:   end;
22:
23: procedure MasKlass.StellZahl(Index:integer;G:integer);
24: begin
25:   if index>n then //если индекс вышел за границы массива
26:     exit;
27:   if G>=0 then
28:     R[Index]:=G
29:   else
30:     R[Index]:=0;
31: end;
32:
33: function MasKlass.GibZahl(Index:integer):integer;
34: begin
35:   if index > n then
36:     result:=0
37:   else
38:     result:=R[index];
39: end;
40:
41: function MasKlass.ZurZeile:string;
42: var
43:   i:integer;
44: begin
45:   Result:='';
46:   for i:=1 to n do
47:     Result:=Result+IntToStr(R[i])+' ';
48: end;
49:
50: var
51:   MK:MasKlass;
52:   i:integer;
53: begin
54:   MK:=MasKlass.Create;
55:   for i:=1 to n do
56:     MK.Mass[i]:=i - 2;
```

```
57:   writeln(MK.ZurZeile);
58:   readln;
59: end.
```

Зачем вообще нужны свойства? Если мы хотим получить доступ к полю, достаточно написать 2 метода. Зачем еще вводить свойство, которое бы вызывало эти 2 метода? Один из возможных ответов – в удобстве для пользователя класса: не надо запоминать 2 метода, а достаточно лишь одно свойство. Кроме того, иногда можно сэкономить на методах, ссылаясь непосредственно на поле.

Но этих доводов мало для того, чтобы вводить свойства. Принципиальнее то, что свойства вводят как бы внутреннюю регуляцию у объектов класса: мы передаем информацию объекту, а он сам распоряжается ею (подключает собственные методы). К вопросу о полезности или бессмысленности свойств мы еще вернемся в последней главе этой книги. Там же мы рассмотрим иной способ введения саморегуляции, который будет гораздо более общим, чем использование свойств.

Так или иначе, но свойства активно используются в встроенной библиотеке классов Delphi, поэтому знать как они работают важно для программирования в среде Delphi.

16.18. Взаимодействие классов

Часто случается ситуация, когда первый класс использует второй как параметр одной из функций, а второй класс использует аналогичным образом первый класс.
Простейший пример приведен ниже:

Пример 16: Зацикливание при обращении к классам.

```
{$APPTYPE CONSOLE}

uses
  SysUtils;
type
  Kl1 = class
    procedure P1(Obj:Kl2);
  end;
  Kl2 = class
    procedure P2(Obj:Kl1);
  end;

procedure Kl1.P1(Obj:Kl2);
begin
end;

procedure Kl2.P2(Obj:Kl1);
begin
end;

begin
end.
```

Хотя мы можем создавать лишь ссылки на объекты, но при этом компилятор все равно хочет, чтобы они не ссылались одна на другую. Чтобы такую зависимость сделать возможной, в Delphi можно делать предварительное объявление класса. Так можно исправить ошибки в предыдущем примере:

Пример 17: Обход зацикливания.

```
{$APPTYPE CONSOLE}
uses
  SysUtils;
type
  KL2 = class;  //предварительное объявление
  KL1 = class
    procedure P1(Obj:KL2);
  end;
  KL2 = class
    procedure P2(Obj:KL1);
  end;

procedure Kl1.P1(Obj:Kl2);
begin
end;

procedure Kl2.P2(Obj:Kl1);
begin
end;

begin
end.
```

До объявления класса KL1 надо сделать предварительное объявление класса KL2. Для этого достаточно написать строку:

```
Имя_Класса = class;
```

Только не перепутайте с объявлением нового класса:

```
Имя_Класса = class
end;
```

16.19. Интерфейсы

Мы с вами уже знаем записи, которые представляют собой набор данных и классы – данные и функции «в одном флаконе». Не хватало лишь набора функций. Для симметрии решили добавить и такой тип данных, и назвали его интерфейсом. Кроме методов могут быть в интерфейсе и свойства, но ссылаться они должны, естественно, лишь на методы интерфейса.

Вот пример интерфейса:

```
MeinInterf = interface (IInterface)
   function ZurZeile:string;
   function MeineFunk(x,y:real):real;
end;
```

Как вы уже догадались, у интерфейсов тоже есть понятие наследования. Базовый интерфейс, от которого всегда наследует любой другой интерфейс – это IInterface.

Функции, которые описаны в интерфейсе, не определяются, как и абстрактные функции. Фактически, интерфейс – это аналог абстрактного класса, только в нем все без исключения функции являются абстрактными.

Интерфейсы используются для создания на их основе новых классов: Класс может наследовать методы от интерфейса (при этом говорится, что класс удовлетворяет интерфейсу). Тогда в классе должны быть написаны реализации для всех методов, которые перечислены в объявлении интерфейса.

Пример объявления класса:

```
Interfklasse = class (MeinInterf)
        {объявления полей, методов и свойств класса}
public
   function ZurZeile:string;
   function MeineFunk(x,y:real):real;
end
```

Пример 18: Построение класса с использованием интерфейсов.

```
program Interrf1;

{$APPTYPE CONSOLE}

uses
  SysUtils;

type
  ISchreib = interface(IInterface)
    function ZurZeile:string;
  end;

  SchrKl = class(TInterfacedObject,ISchreib)
  private
    x:integer;
  public
    constructor Bilde(y:integer);
    function ZurZeile:string;
  end;

constructor SchrKl.Bilde(y:integer);
begin
  x:=y;
end;

function SchrKl.ZurZeile:string;
begin
  result:=IntToStr(x);
end;

var
  S:SchrKl;
```

```
begin
  S:=SchrKl.Bilde(23);
  writeln(S.ZurZeile);
  readln;
end.
```

Наследовать класс может только от одного класса, но он может удовлетворять нескольким интерфейсам (они должны перечисляться через запятую в скобках после слова class).

С помощью интерфейсов можно вводить родство между классами по функциональному наполнению. Это бывает полезно, т.к. некоторые действия различных классов могут быть очень похожими – следовательно, можно сделать интерфейс, которому удовлетворяли бы оба класса. Например, глаза у осьминога и человека очень похожи по строению, хотя ветви дерева видов, которым принадлежат человек и осьминог, разошлись очень давно.

В Delphi можно динамически проверять принадлежность класса интерфейсу с помощью оператора as. Но для этого интерфейс должен удовлетворять следующим условиям:

1. Интерфейс должен удовлетворять интерфейсу IInterface (т.е. этот интерфейс должен обязательно стоять в списке интерфейсов в скобках после ключевого слова interface).
2. У интерфейса должен быть специальный номер – GUID. Он следует после списка интерфейсов, от которых наследует интерфейс.

Этот номер нужен для того, чтобы каждый интерфейс (т.е. набор функций) можно было бы идентифицировать независимо от того, какое у него название. Чтобы создать GUID в Delphi можно просто нажать комбинацию клавиш Ctrl+Shift+G. Тогда будет вызвана специальная функция, генерирующая номер.

В следующем примере создается массив ссылок на объекты типа TInterfacedObject. Этот массив мы будем заполнять случайным образом ссылками на объекты 3-х разных классов, два из которых удовлетворяют интерфейсу IEtw, а один - нет. Потом мы должны будем пройтись по элементам массива и для каждого элемента проверить, удовлетворяет ли он интерфейсу IEtw, и если да, то вызвать метод, который описан в интерфейсе.

Для динамической проверки, удовлетворяет ли объект интерфейсу, используется оператор as (см. строку 58):

```
V:=M[i] as Ietw;
```

Если M[i] удовлетворяет интерфейсу IEtw, то в V будет записана ссылка на интерфейс. В противном случае возникнет исключительная ситуация. Подробнее мы рассмотрим обработку исключительных ситуаций в следующей главе, пока же остановимся на главном: если возникает исключительная ситуация во время работы программы (т.е. выполняется некоторое недопустимое действие), то можно с помощью специального оператора продолжить выполнение программы. Для этого служит оператор try, после которого идут операторы, которые могут быть потенциально опасны. После этих операторов можно поместить секцию except, которая предписывает действия, которые надо выполнить в случае, если при выполнении этих операторов возникла ошибка.

Пример 19: Проверка, удовлетворяет ли класс интерфейсу.

```
1 : program InterfAs;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils;
7 :
8 : const
9 :   n=10;
10: type
11:   IEtw = interface(IInterface)['{D3835685-687C-44EF-A6D6-
FCE01886C091}']
12:     function Neues:integer;
13:   end;
14:
15:   Kl = class(TInterfacedObject,IEtw)
16:     function Neues:integer;
17:   end;
18:
19:   Kl2 = class(TInterfacedObject,IEtw)
20:     function Neues:integer;
21:   end;
22:
23:   Kl3 = class(TInterfacedObject)     //не поддерживает интерфейс
24:   end;
25:
26:   Mas = array [1..n] of TInterfacedObject;
27:
28: function Kl.Neues:integer;
29: begin
30:   result:=5;
31: end;
32:
33: function Kl2.Neues:integer;
34: begin
35:   result:=6;
36: end;
37:
38: var
39:   V:IEtw;
40:   M:Mas;
41:   i,x:integer;
42: begin
43:   randomize;
44:   for i:=1 to n do
45:     begin
46:     x:=random(3);
47:     case x of
48:     0: M[i]:=Kl.Create;
49:     1: M[i]:=Kl2.Create;
50:     2: M[i]:=Kl3.Create;
```

```
51:     end
52:     end;
53:
54:   for i:=1 to n do
55:     begin
56:     try
57:       write(M[i].ClassName,' ');  //печатаем название класса
58:       V:=M[i] as Ietw;
59:       writeln(V.Neues);
60:     except
61:       writeln;  //ничего не делаем
62:     end;
63:     end;
64:
65:   readln;
66: end.
```

Окончательный анализ концепций ООП мы дадим в заключительной части этой книги. Сейчас мы ответим на один из вопросов, который был сформулирован в части «Выведение», а именно: каким образом можно избежать применения бестиповых указателей и при этом достичь общности программ? Решение этой проблемы достигается с помощью следующих нововведений:

1. Используются только ссылки на объекты.
2. Все объекты имеют общего родителя – TObject
3. Контроль типов во время выполнения (механизм RTTI)

Кроме того, можно использовать метаклассы, что дает дополнительные возможности для написания абстрактных программ.

16.20. Новое в Delphi 2005

Все, о чем шла речь в этой главе, было еще в 6-й версии Delphi. Сейчас мы очень кратко рассмотрим новинки, которые появились в Delphi 2005. Сразу скажу, что с точки зрения языка все нововведения не являются фундаментальными. Большинство из них – лишь желание подстроиться под платформу .NET. Некоторые из новинок можно использовать только работая на .NET, а при работе в приложениях под Win32 их нет. Из-за этого программисту нужно запоминать много лишней информации, да и красота языка страдает.

Определители видимости элементов класса

В Delphi 2005 появилось 2 новых определителя видимости элементов класса – strict private и strict protected. Смысл в них такой: если private-элементы видны не только внутри класса, но и внутри модуля, в котором объявлен класс, то strict private-элементы видны только внутри класса. Аналогично, strict protected-элементы класса видны лишь внутри класса и его потомков.

В принципе, введение подобных дескрипторов правильно, ибо неясно, зачем разрешать доступ к членам класса его соседям по модулю. Другое дело, зачем вообще было вводить обычные private и protected?

Запечатанные и абстрактные классы

В Delphi 2005 появились 2 новых вида классов: sealed (запечатанные), abstract (абстрактные). У запечатанных классов не может быть наследников, но можно создавать экземпляры класса, а у абстрактных классов наоборот: нельзя создавать экземпляры классов, но можно создавать потомков. У абстрактного класса не обязательно должны быть абстрактные функции, а класс с абстрактными методами не обязан быть абстрактным (абстрактность методов и абстрактность классов – независимые понятия)[16].

Запечатанность классов действует и при программировании на .NET и при использовании Win32. А абстрактность классов имеет смысл лишь в .NET (при программировании приложений под Win32 слово abstract писать можно, но оно не будет ничего означать).

В следующем примере (под Win32) если раскомментировать определение класса R, то будет выдана ошибка, т.к. от класса, помеченного sealed наследовать нельзя, но при этом вполне можно создавать экземпляры абстрактных классов.

Пример 20: Запечатанные и абстрактные классы.

```
program SealAbstr;
{$APPTYPE CONSOLE}
uses
  SysUtils;
type
  Sld = class sealed
    x:integer;
  end;
  Abst = class abstract
  x:integer;
  end;
{  R = class (Sld)  //Ошибка - нельзЯ наследовать
  end;}
var
  A:Abst;
begin
  A:=Abst.Create; {Ошибки нет!!!}
  A.x:=6;
  writeln(A.x);
  readln;
end.
```

Вложенные классы

В Delphi 2005 разрешили внутри класса объявлять вложенные подклассы. Тогда они не видны вне этого класса (хотя мы в примере посмотрим, что на самом деле можно добраться и до них). Рассмотрим на примере, как определяются вложенные классы.

---

[16] Классы, которые мы раньше называли абстрактными, в Delphi 2005 логичнее называть, чтобы не возникало путаницы, абстрактнофункциональными.

Пример 21: Использование вложенных классов.

```
1 : program InnKlassen;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils,
7 :   Graphics;
8 :
9 : type
10:   Pixel = class
11:   type
12:     Punkt = class       {вложенный класс}
13:     strict private
14:       x,y:real;
15:     public
16:       constructor Bilde(xx,yy:real);
17:       function GibEukNorm:real;
18:     end;
19:
20:   strict private
21:     P:Punkt;
22:     F:TColor;
23:   public
24:     constructor Bilde(x,y:real;frb:TColor);
25:   end;
26:
27: constructor Pixel.Punkt.Bilde(xx,yy:real); //просто Punkt.Bilde
писать нельзЯ
28: begin
29:   x:=xx;
30:   y:=yy;
31: end;
32:
33: function Pixel.Punkt.GibEukNorm:real; //просто Punkt.GibEukNorm
писать нельзЯ
34: begin
35:   result:=sqrt(x*x+y*y);
36: end;
37:
38: constructor Pixel.Bilde(x,y:real;frb:TColor);
39: begin
40:   if x<=0 then
41:     x:=0;
42:   if y<=0 then
43:     y:=0;
44:   P:=Punkt.Bilde(x,y);
45:   F:=frb;
46: end;
47:
48: var
```

```
49:   Pix:Pixel;
50:   P:Pixel.Punkt;  {Написать просто P:Punkt нельзЯ}
51: begin
52:   Pix:=Pixel.Bilde(10,20,$00ff0000);
53:   P:=Pixel.Punkt.Bilde(10,-10);
54:   writeln(P.GibEukNorm:8:6);
55:   readln;
56: end.
```

Внутри класса Pixel объявляется вложенный класс Punkt. Сам же класс Pixel содержит объект класса Punkt в качестве строго закрытого поля. Чтобы написать реализации для методов вложенных классов, надо (см. строки 27, 33), надо указать имя внешнего класса, затем – имя внутреннего и потом – название метода.

Если попытаться в разделе var указать P:Punkt (строка 50), то компилятор выдаст сообщение об ошибке (класс-то вложенный), но тем не менее разработчики Delphi дали возможность программисту все же объявить экземпляры вложенный классов вне класса, в котором он объявлен (см. стр. 50). Быть может, разработчики Delphi решили ввести более громоздкую запись для обозначения вложенных классов из-за того, что в двух разных классах могут быть вложенные классы с одинаковым названием (но разной реализацией). Мне кажется, что логичнее было бы сделать вложенные классы вообще недоступными вне класса-оболочки.

Пространства имен

В TP и Delphi программный код разбивался на модули, а каждый модуль был своеобразной оболочкой для класса. В Delphi 2005 в дополнение к модулям добавили понятие пространства имен. С точки зрения Delphi пространства имен – это надмодульные образования, которые позволяют строить иерархии. Пространства имен ввели из-за того, что количество классов, которые используются в программах, а также количество модулей, стало огромным (количество классов стандартных библиотек для студий программирования уже исчисляется тысячами). Поэтому хотелось каким-то образом навести порядок, тем более что иногда встречаются одинаковые имена классов и надо как-то их различать. Мы напишем один элементарный пример, на котором мы рассмотрим, как объявляются пространства имен. Более детальную информацию вы можете получить в справочной системе.

Пример 22: Использование пространств имен.

--------------Модуль Unit1 пространства имен MeineNamespace-----------

```
1 : unit MeineNamespace.Unit1;
2 :
3 : interface
4 : type
5 :   Klasse = class
6 :   end;
7 :
8 : implementation
9: end.
```

--------------Модуль Unit2 пространства имен MeineNamespace-----------

```
1 : unit MeineNamespace.Unit2;
```

```
2 :
3 : interface
4 : type
5 :   Klasse = class
6 :   private
7 :     x:integer;
8 :   end;
9 :
10: implementation
11: end.
```

--------------Основной файл проекта-----------

```
1 : program NameSp;
2 : {$APPTYPE CONSOLE}
3 :
4 : uses
5 :   MeineNamespace.Unit1,
6 :   MeineNamespace.Unit2;
7 : var
8 :   Kl:MeineNamespace.Unit1.Klasse;
9 :   Kl2:Klasse;
10: begin
11:   Kl:=MeineNamespace.Unit1.Klasse.Create;
12:   Kl2:=Klasse.Create;
13: end.
```

Чтобы вложить модуль в пространство имен, надо перед названием модуля поставить название соответствующего пространства имен и поставить точку. Файл с кодом модуля (для MeineNamespace.Unit1) будет называться при этом (MeineNamespace.Unit1.pas).

В примере 22 в обоих модулях определен класс Klasse. Для того чтобы объявить экземпляр нужного класса Klasse, надо указать модуль, в котором он находится (см. строку 8). В строке 9, несмотря на то, что мы пишем просто

```
Kl2:Klasse;
```

имеется в виду

```
Kl2: MeineNamespace.Unit1.Klasse;
```

Еще кое-что

В Delphi 2005, если программировать на платформе .NET, можно использовать т.н. перегрузку операторов. Перегрузка операторов для классов – это возможность использования операций и операторов для экземпляров класса. Например, можно сделать класс «длинное целое», для которого перегрузить все стандартные операции (арифметические и логические операции). Перегрузка операторов (и операций) действительно выглядит симпатично, когда классы – это некоторые математические объекты, для которых ясно, какой смысл вкладывается в операции. Но мне кажется, что этих преимуществ недостаточно, чтобы оправдать введение перегрузки операторов.

Быть может, оригинальнее было бы функции для двух аргументов записывать не «Funk(a,b)», а «a Funk b». В таком случае можно было бы интерпретировать операции как частный случай функций.

Есть в Delphi 2005 понятие помощников классов. С их помощью можно расширять классы, не создавая наследников классов. Часто пишут, что helper'ы очень полезны тогда, когда требуется расширить классы, которые помечены как sealed. Но в таком случае, зачем было вообще вводить sealed-классы, чтобы потом давать окольные пути для их расширения.

В общем, все нововведения Delphi 2005 выглядят очень спорными: такое ощущение, что они созданы скорее для того, чтобы сделать язык более громоздким, чем чтобы сделать его красивее и мощнее.

Задачи

1. Создайте класс Mensch (человек), в котором бы хранились сведения о человеке: фамилия, имя, отчество, телефон, адрес, e-mail. Вы должны написать базовые методы класса: возможность установки/чтения полей, преобразование в строку и т.д.
2. Создайте наследника класса Mensch – Student, в котором добавились бы следующие поля:
   - ВУЗ, в котором учится студент
   - Стипендия
3. Объявите интерфейс KannVergleichen ("могу сравнивать"), в котором был бы метод, позволяющий сравнить 2 элемента типа Mensch.
4. Напишите класс, который представлял бы собой список ссылок на объекты класса Mensch. Этот класс должен удовлетворять интерфейсу KannVergeichen. Реализуйте основные операции над списком (вставка, элементов в начало и конец списка, вставка элемента в произвольную позицию и удаление элемента), а также сортировку с помощью метода, который объявлен в интерфейсе.
5. Добавьте в класс, который вы написали в предыдущем примере метод поиска заданного человека.
6. Напишите класс Arithmetik, в котором были бы следующие методы класса:
   - Возведение числа в целую степень (количество операций должно быть порядка логарифма от показателя степени)
   - Возведение числа в степень по заданному модулю (т.е. вычисление $a^b \bmod n$)
   - Нахождение НОД двух чисел
   - Вычисление функции Эйлера $\varphi(n)$ ($\varphi(n)$ = количеству простых делителей числа n).
7. Создайте класс-обертку Menge динамического массива, который имитировал бы множество (т.е. фактически мы будем иметь дело с битовым массивом).
8. Добавьте в класс Arithmetik (см. упражнение 6) функцию, которая вычисляет количество целых чисел от 0 до n включительно с помощью перебора делителей для каждого числа.
9. Добавьте в класс Arithmetik (см. упражнение 6) функцию, которая вычисляет количество целых чисел от 0 до n включительно с помощью решета Эратосфена (используйте класс Menge из упражнения 7).
10. Подумайте, каким образом могут быть реализованы наследование и полиморфизм с помощью чудо-записей.

Проект 5: Криптография

Основные определения:

- Криптография – это наука, изучающая шифры, методы шифрования и расшифровки информации.
- Шифрование – преобразование текста по некоторому алгоритму.
- Ключ – параметр алгоритма шифрования.

Нашей целью будет написать программу, которая будет зашифровывать сообщения, закодированные простыми методами и взламывать их. Это – основная цель. Но вы помните, что большие проекты должны удовлетворять и другим качествам, а именно: возможность модернизации программы, а также абстрактность (т.е. возможность использования кусков программы для написания ПО, имеющего достаточно отдаленное отношение к исходной программе. Поэтому вы должны писать классы, которые можно будет использовать не только для криптографии, но и для задач, связанных с лингвистикой.

Шифры применялись еще в древности: многие переписки государственных деятелей зашифровывались, чтобы обеспечить сохранность информации.

Рассмотрим 2 простых шифра.

1. Шифр Юлия Цезаря

Пусть буквы пронумерованы от 0 до n-1. Тогда согласно правилу Цезаря буква с номером x переходит в букву с номером $(x+a)\bmod n$.

Сдвиг на 4 единицы можно представить следующей подстановкой: (A[1,j] ->A[2,j]).

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	a	b	c	d

С помощью этого шифра сообщение

I am programmer. превратится в M eq tvskveqqiv.

2. Подстановочный шифр

Подстановочный шифр – обобщение шифра Цезаря: в подстановке нижняя строка – произвольная перестановка букв латинского алфавита.

Расшифровка текстов

Если ключ известен, то расшифровать текст легко. Если же вы ключ не знаете, то сложностей гораздо больше.

Для расшифровки шифра Цезаря достаточно знать одну букву, - после этого находим смещение, и легко восстанавливаем весь недостающий алфавит.

Поэтому наиболее простой метод расшифровки этого шифра – просто перебрать все 26 возможных ключей, и посмотреть, какой из них подходит к тексту (т.е. слова, получившиеся после подстановки шифра получились правильными).

Ясно, что для этого надо запастись хорошим словарем.

К сожалению, этот метод не подходит для расшифровки подстановочного шифра – в нем количество возможных вариантов = 26!, и перебор, естественно, невозможен.

Для расшифровки этого шифра надо использовать информацию о частотности букв, буквосочетаний и слов в предложении, а также различные дополнительные сведения из грамматики.

1. Частота встречаемости букв
2. Знания о структуре слова
   - Наиболее часто используемые приставки, суффиксы и окончания
   - Часто используемые буквосочетания (например, сдвоенные буквы)
3. Частота встречаемости слов
   - Короткие слова
   - Избыточность языка (напр., артикли)
4. Структура предложения
   - Порядок слов в предложении
   - Стандартные конструкции (напр., вспомогательные глаголы при образовании времен)
   - Апострофы (после них или перед ними во многих языках могут стоять лишь некоторые буквы)
5. Осмысленность предложения

Разумеется, вы можете предлагать и собственные источники знаний.

Сам алгоритм расшифровки – перебор с возвратами. Надо собирать информацию от всех источников знаний, и делать предположения о возможных подстановках. Если предположение неверно, то надо возвращаться на шаг назад (или сразу на несколько шагов).

Пока у вас не будет написано хотя бы несколько источников знаний, алгоритм писать бессмысленно. Поэтому первоочередной задачей для вас будет научиться строить словарь всех слов, используемых в тексте и вычислять частотность букв (знания о структуре предложения – сложнее по своей сути, а осмысленность требует от ПК интеллектуальных способностей).

Для написания программы используйте принципы ООП. Можно написать классы, содержащие знания о частотности букв, слов и т.д., а также класс-расшифровщик, который будет использовать источники знаний.

Но не забывайте, что ваша программа должна быть адаптирована для работы с любым языком, поэтому вы должны писать программу так, чтобы на более абстрактных уровнях она не использовала знаний о конкретных языках.

Некоторые подзадачи:

1. Напишите класс, который генерирует ключ, а также может зашифровывать и расшифровывать (по известному ключу) тексты, содержащиеся в файлах.
2. Источник знаний для работы со словами должен уметь:
   - Составлять список слов в тексте (разные словоформы одного слова считаются разными словами) вместе с их частотой встречаемости.
   - Сортировать этот список (список будет большой, поэтому алгоритм должен быть быстрый).
   - Искать в словаре заданное слово, и выдавать частоту встречаемости данного слова.

- Поиск по маске

Маска:

- Любой элемент **- ?**
- Не элемент **- ~**
- Несколько элементов **- ***
- Начало группы **- {**
- Конец группы **- }**

Например, шаблон «?{a b d}?a~{s d e}» означает, что первая буква слова неизвестна, вторая – a, b или d, третья – неизвестна, четвертая – a, пятая – любая буква, не принадлежащая множеству {s, d, e}.

3. Главный дешифровщик должен использовать данные от всех других источников знаний, строить предположения о буквах и если они неверны, то возвращаться на несколько шагов назад.

Глава 17: Визуальное программирование

Вы уже наверняка запустили стандартное оконное приложение, которое генерирует Delphi, и, быть может, перетащили на него несколько кнопок или текстовых полей. Теперь пришло время разобраться, как функционируют оконные приложения и как на них реагирует ОС.

17.1. Оконные приложения в Windows

Операционные системы могут полностью скрывать аппаратуру, либо напротив, предоставлять программам возможность прямого доступа к аппаратному обеспечению (АО). Windows XP полностью скрывает АО, а Windows 95 и 98 – дают возможность прямого доступа к АО. В дальнейшем будем рассматривать ОС, которые скрывают аппаратуру.

Если мы хотим что-то нарисовать на экране монитора, то сделать это напрямую, послав монитору некоторую последовательность команд, нельзя. Для этого Windows предоставляет набор функций, которые позволяют рисовать на экране отдельные пиксели (мельчайшие единицы изображения на экране монитора; разрешение монитора показывает количество пикселей по горизонтали и вертикали монитора) и простые фигуры (линии, многоугольники, эллипсы и т.д.).

То, что мы можем с помощью Win32 API-функций работать с устройствами – это понятно, однако часто надо, чтобы сигналы от устройств влияли на работу приложений. Простейший пример этому – щелчок на кнопке. Раньше мы никогда не сталкивались с подобной ситуацией: мы могли сказать пользователю ввести некоторые данные, но если мы его не просим, то он может нажимать на кнопки сколько угодно – толку от этого все равно никакого. Зависимость приложения от сигналов, посылаемых внешними устройствами может быть реализована лишь с помощью ОС (т.к. она берет на себя всю работу с аппаратурой).

Упрощенная схема того, каким образом Windows обрабатывает сигналы от внешних устройств, выглядит так:

Когда какое-то устройство, например, мышь, посылает сигнал в ОС (например, что она переместилась вправо на столько-то пикселей), то ОС принимает этот сигнал и создает **сообщение**, которое она посылает приложению. В оконном приложении есть специальная функция («оконная функция»), которая может принимать эти сообщения и, при надобности, их обрабатывать. Например, если пользователь щелкнул левой кнопкой мышки в области окна, мышь сообщит ОС о том, что была нажата ее левая кнопка, а та вышлет сообщение WM_LEFTBUTTONDOWN, которое говорит приложению, содержащему это окно, о том, что была нажата левая кнопка мыши, а приложение уже само разбирается, какой смысл этому надо придать.

- Многие важные сообщения, посылаемые ОС приложению, приводят к **событиям** – реакциям на сообщения ОС.
- Если программное обеспечение (ПО) построено таким образом, что на его работу могут влиять события, то такое ПО называется событийно-управляемым.

Для того, чтобы упростить разработку визуальных приложений, добиться единства интерфейса в прикладных программах и ускорить работу самих приложений, в Windows разработан GUI (Graphic User Interface) – набор подпрограмм, которые

позволяют создавать окна, кнопки, меню, списки и т.д. Все эти подпрограммы тоже входят в состав Win32 API.

17.2. Создание оконных приложений в Delphi

В Delphi создавать оконные приложения очень просто. Во-первых, у каждого оконного приложения есть специальный объект Application (англ. приложение) класса TApplication, который берет на себя обработку сообщений от Windows. У этого класса есть метод Run, основную часть которого составляет цикл, в котором он обрабатывает сообщения Windows.

Во-вторых, встроенная среда разработки показывает список компонентов (**компонент** – это любой наследник класса TComponent стандартной библиотеки классов), которые можно просто перетаскивать на форму (окно).

У компонентов есть набор событий, на которые они могут реагировать.

- Метод, вызывающийся при возникновении события, называется обработчиком события.

Основным компонентом является форма, на которой размещаются остальные компоненты. Форма – это любой наследник класса TForm. Сам класс TForm инкапсулирует все необходимое (с точки зрения разработчиков Delphi) для работы с пустым окном – его можно перемещать, изменять размеры, сворачивать, восстанавливать и т.д.

17.3. Первое оконное приложение

Теперь создайте приложение с формой (при создании нового приложения выберите VCL Forms Application для Win32 (в версиях Delphi 6,7 – просто Application) , а не Console Application, как мы делали ранее).

Давайте разберемся с программным кодом. Нажав на Code, вы увидите вспомогательный модуль. Для того, чтобы открыть основную программу и остальные файлы с программным кодом, связанные с вашим проектом (так называется приложение в Delphi), вы можете просто нажать Ctrl+F12 (или нажать соответствующую кнопку – она находится слева от кнопки запуска программы.

Нажмите Ctrl+F12, и откройте файл Project1 (так по умолчанию называется проект). Это и есть основная программа.

Итак, простейшая программа состоит из 2-х файлов: основного и вспомогательного модуля, в котором находится форма.

Пример 1: Простейшая программа.

```
------------------- файл Project1.dpr -----------------------
1 : program Project1;
2 :
3 : uses
4 :   Forms,
5 :   Unit1 in 'Unit1.pas' {Form1};
6 :
7 : {$R *.res}
8 :
```

```
9 : begin
10:   Application.Initialize;
11:   Application.CreateForm(TForm1, Form1);
12:   Application.Run;
13: end.
```

```
---------------------- файл Unit1.pas --------------------------
1 : unit Unit1;
2 :
3 : interface
4 :
5 : uses
6 :   Windows, Messages, SysUtils, Variants, Classes, Graphics,
7 :   Controls, Forms, Dialogs;
8 :
9 : type
10:   TForm1 = class(TForm)
11:   private
12:     { Private declarations }
13:   public
14:     { Public declarations }
15:   end;
16:
17: var
18:   Form1: TForm1;
19:
20: implementation
21:
22: {$R *.dfm}
23:
24: end.
```

Теперь я предлагаю вам запустить программу. Вы видите что, не написав ни строчки кода, у вас уже есть готовое окошко, причем вы можете изменять размеры окна, сворачивать и закрывать его. Вся эта функциональность уже заложена в классе TForm.

Во вспомогательном модуле описан один класс – TForm1, являющийся наследником TForm (правда, ничем не отличающимся от своего родителя), а также объект Form1 этого класса.

С помощью директивы `{$R *.dfm}` подключаются файлы, связанные с формами. В основном файле проекта с помощью

```
{$R *.res}
```

подключаются файлы ресурсов, связанные с проектом.

В трех строках в инициализирующей части вызываются три метода для глобального объекта Application класса TApplication. Метод Initialize выполняет вспомогательные функции, метод CreateForm инициализирует форму начальными значениями (в этот метод передается класс, которому принадлежит форма и сам объект-форма, следовательно, первый параметр метода CreateForm класса Application – это метакласс).

Метод Run запускает оконное приложение.

Форма должна быть в находиться в отдельном pas-файле. В придачу к этому pas-файлу будет создан одноименный файл с расширением dfm, в котором будут храниться параметры настройки формы.

17.4. Добавляем компоненты на форму (пример № 2)

Создайте новое приложение (VCL Forms Application), затем найдите в наборе компонентов во вкладке Standard следующие 2: текстовое поле (TMemo) и кнопку (TButton).

Щелкнув по компоненту дважды, вы добавите его на форму. После этого его можно перемещать по форме и изменять его размеры.

Создайте форму, как показано на рис. 17.1.

В программном коде у формы добавится 2 поля:

```
TForm1 = class(TForm)
    Button1: TButton;
    Memo1: TMemo;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

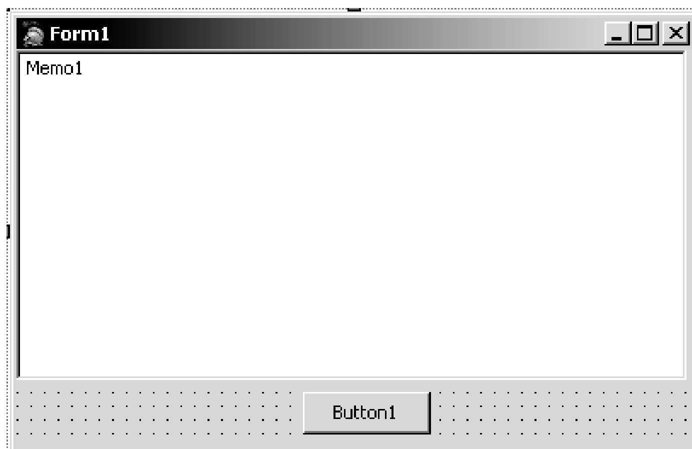


Рис 17.1 Форма для примера № 2

Запустите теперь программу и посмотрите, что теперь можно с ней делать.
Вы можете писать что-то в текстовом поле; можно нажимать на кнопку, хотя при этом ничего не происходит.

Кнопка и текстовое поле содержат ряд свойств, значения которых можно изменять. Самый простой способ сделать это – использовать Object-Inspector.

На рис. 17.2 изображен инспектор объектов (ИО) с краткими объяснениями. Вы можете выбрать в инспекторе объектов интересующий вас компонент и вы увидите свойства, которые есть у этого объекта. После этого просто в правой половине таблицы вы можете изменять значения этих свойств.

Измените заголовок формы (в ИО он называется Caption). Измените название на любое, которое вам нравится.

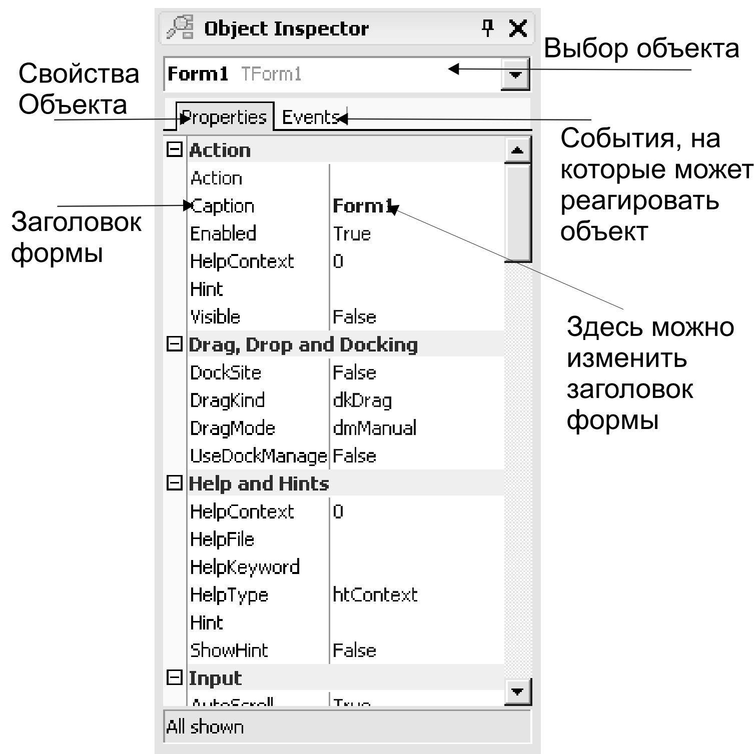


Рис. 17.2 Инспектор объектов

Теперь в качестве объекта выберите кнопку. Измените у нее название (Name) на Knopf. Можете изменить и название текстового поля с Memo1 на иное.

Давайте изменим начальное содержимое текстового поля. Для этого надо сделать следующие действия: найдите свойство Lines класса TStrings у объекта Memo1:

Теперь нажмите на маленькую кнопку с тремя точками. В окне, которое появится на экране вы сможете изменить текст.

Запустите программу, чтобы убедиться, что вы все сделали верно.

Напоследок мы научимся делать некоторые действия при нажатии на кнопку. Для этого дважды щелкните по кнопке. Оболочка Delphi сразу же добавит к форме следующий метод:

```
procedure TForm1.KnopfClick(Sender: TObject);
begin

end;
```

У этого метода лишь 1 параметр, - объект, который отослал сообщение о событии. В данном случае Sender – сама кнопка.

Внутри этой процедуры допишите 1 строку, чтобы получилось следующее:

```
procedure TForm1.KnopfClick(Sender: TObject);
begin
   Form1.TextPlatz.Lines.Add('Вы нажали кнопку');
end;
```

TextPlatz – название текстового поля (если вы придумали другое название, то поставьте его вместо TextPlatz).

Внутри TextPlatz есть свойство Lines, в котором есть метод Add, которая добавляет строку, которая передана этой функции в текстовое поле. Поэтому строка `'Вы нажали кнопку'` будет дописываться в текстовое поле после каждого нажатия на кнопку Knopf.

Как видите, мы написали лишь одну строчку кода, а наша программа уже содержит кнопки, текстовые поля, и т.д. Это, конечно, не идет ни в какое сравнение с ТР, в котором надо было практически все делать самому. Но такое «программирование интерфейса», блуждания по меню и установка начальных свойств – дело необычайно нудное и глупое. Поэтому вы можете возмутиться тем, что после стольких проектов вы должны заниматься такой ерундой:

Тяжелая доля
досталась герою;
вождю довелось
зерна молоть!
Руке той привычна
меча рукоять,
а вовсе не палка,
что жернов вращает.

Старшая Эдда. Вторая песнь о Хельги убийце Хундинга

Однако написание всех методов «руками» - тоже занятие весьма глупое, поэтому почему бы не сэкономить себе время там, где это можно, а затем заняться более подобающим делом, как и Хельги, который славными подвигами покрыл свое имя, и после смерти стал править Вальхаллой вместе с Одином.

Теперь сохраните программу, которую вы создали. Я вам советую для каждой программы создавать отдельный каталог. Иначе вы быстро запутаетесь в ваших программах.

17.5. Убегающая кнопка. Обработка события OnMouseMove (пример № 3)

Некоторые компоненты Delphi (в частности, кнопки и текстовые поля, могут реагировать на события. Количество событий, на которые могут реагировать компоненты, различно. Для каждого компонента их список перечислен в Object Inspector, вкладка Events.

Нажатие на кнопку – это событие OnClick. Процедура, в которой задаются действия в ответ на событие, называется **обработчиком события**. В предыдущей программе мы научились создавать обработчик события OnClick. Для этого мы 2 раза нажимали на кнопке, и система самостоятельно создавала заголовок обработчика сообщения. Если вы теперь откроете тот проект, и для объекта Knopf откроете в Object Inspector вкладку Events, то вы увидите, что в поле OnClick стоит имя процедуры, которая была создана системой разработки.

В следующем примере мы с вами напишем программу, которая будет интересоваться, нравится ли вам ваша жизнь. Причем, как только вы хотите нажать кнопку «Нет», то кнопка будет сдвигаться так, чтобы вы ее не могли нажать. Если же вы хотите нажать кнопку «Да, очень», то никто не помешает вам это сделать. Проект

будет называться ButtBeweg. Советую проделать вам все действия по созданию самой формы самостоятельно, а лишь затем запускать готовый проект.

Форма должна иметь следующий вид (шрифт надписей можете выбирать по вашему желанию):

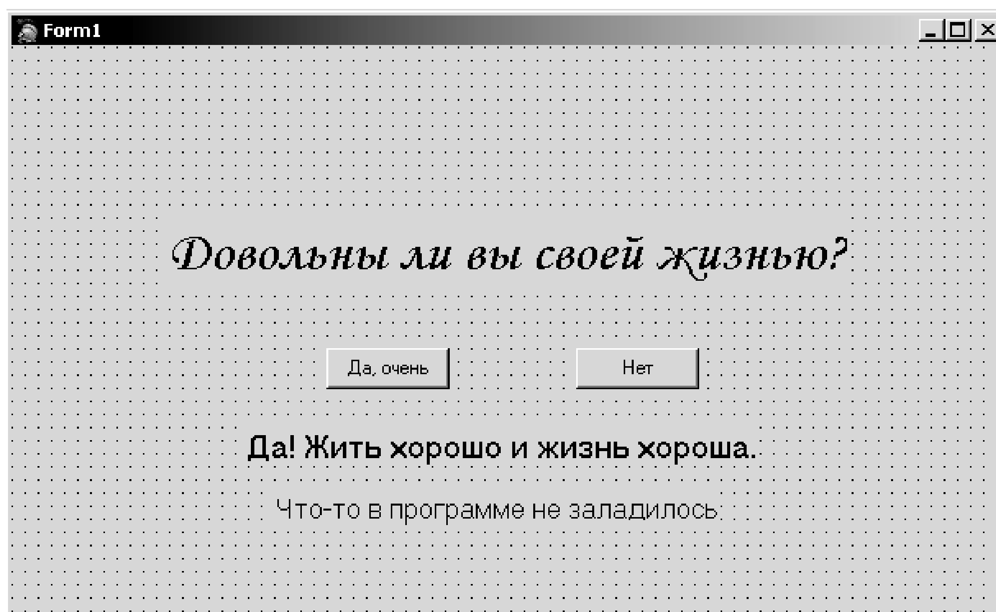


Рис 17.3. Форма для примера №3

В начале работы программы надписи «Да! Жить хорошо и жизнь хороша»и «Что-то в программе не заладилось» должны быть невидимы и появляться лишь после нажатия на соответствующую кнопку.

Для создания надписей служит компонент Label. Создайте 3 таких компонента, а также 2 кнопки. После того, как вы создадите 3 Label, вы увидите, что надо еще изменить сам текст надписи, шрифт и размер. Вы уже знаете, что все эти свойства можно изменить в Object Inspector'е. Надпись – в раделе Caption, а шрифт и его размер можно изменить в свойстве

Нажмите на кнопку с тремя точками и откроется диалог, в котором вы сможете настроить необходимые параметры.

Если вы это сделали, идем дальше.

Надо сделать обе метки (кроме вопроса) невидимыми. Для этого задайте свойствам Visible (англ. видимый) этих двух компонентов значение False.

Теперь надо не дать пользователю нажать кнопку «Нет». Пользователь может нажать кнопку двумя способами:

1. нажав клавишу Enter, когда фокус ввода находится на кнопке. Например, на следующем рисунке фокус ввода находится на кнопке «Да, очень». Для того, чтобы фокус ввода перемещать с одного компонента на другой, можно просто нажимать Tab. Вы можете сами проверить это: надо лишь создать пустую форму, поместить на ней 2 кнопки и запустить программу.

2. с помощью мыши. Для этого надо поместить курсор над кнопкой и затем щелкнуть.

Чтобы не дать пользователю нажать на Enter, можно при запуске фокус ввода установить на кнопке «Да, очень», и запретить перемещать курсор на кнопку «Нет». Этого можно добиться так:

Зайдите в Object Inspector, выберите кнопку «Нет» (я в примере дал ей имя Nein), и найдите 2 свойства: TabOrder и TabStop. TabOrder устанавливает порядок обхода компонентов. Его значение может меняться в следующем диапазоне -1..32767.
-1 – это особый случай, и мы его рассматривать не будем.
0 – это означает, что при запуске фокус ввода будет именно на нем. С помощью Tab мы можем перейти от 0-го к 1-му компоненту и т.д.

Однако если мы установим свойству Tabstop кнопки Nein значение False, то к ней нельзя будет добраться с помощью клавиши Tab.

Теперь самое главное. Надо научиться перемещать кнопку, как только пользователь наведет на нее курсор мыши. Это сделать очень просто, потому что свойства компонентов можно изменять в процессе работы программы. Когда указатель мыши становится на кнопку, возникает событие OnMouseMove. Мы должны его обработать, т.е. описать, что должна делать программа в ответ на его возникновение. Для этого выберите кнопку Nein в Object Inspector'е и зайдите во вкладку Events. Там найдите событие OnMouseMove, и напишите название процедуры-событиеобработчика NeinMove. После этого будет создан пустой обработчик события. Надо лишь записать туда нужный код.

Координаты кнопки, как и других видимых компонентов, определяются следующими свойствами:
(Left, Top) – координаты верхнего левого угла кнопки. Height - высота, Width – ширина. Сам текст обработчика прост. В качестве параметров передаются координаты курсора мыши в тот момент, когда он оказался над кнопкой.

Кроме того, надо создать еще 2 обработчика нажатия кнопок, которые должны будут делать видимыми соответствующие метки.

Пример 3: Убегающая кнопка.

```
1 : unit Haupt;
2 :
3 : interface
4 :
5 : uses
6 :   Windows, Messages, SysUtils, Variants, Classes, Graphics,
7 :   Controls, Forms, Dialogs, StdCtrls;
8 :
9 : type
10:   TForm1 = class(TForm)
11:     Ja: TButton;
12:     Label1: TLabel;
13:     Nein: TButton;
14:     Label2: TLabel;
15:     Label3: TLabel;
16:     procedure JaClick(Sender: TObject);
17:     procedure NeinClick(Sender: TObject);
18:     procedure NeinMove(Sender: TObject; Shift: TShiftState; X,
Y: Integer);
19:   private
```

```
20:     { Private declarations }
21:   public
22:     { Public declarations }
23:   end;
24:
25: var
26:   Form1: TForm1;
27:
28: implementation
29:
30: {$R *.dfm}
31:
32: procedure TForm1.JaClick(Sender: TObject);
33: begin
34:   Label2.Visible:=true;
35: end;
36:
37: procedure TForm1.NeinClick(Sender: TObject);
38: begin
39:   Label3.Visible:=true;
40: end;
41:
42: procedure TForm1.NeinMove(Sender: TObject; Shift: TShiftState;
X,Y: Integer);
43: var
44:   i,n:integer;
45: begin
46:   n:=15;
47:   with Nein do {with работает и для объектов}
48:     begin
49:     if (y<=Height div 2) then {сдвигаемся по вертикали}
50:       begin
51:       for i:=1 to n do
52:       Top:=Top+1;
53:       end
54:     else
55:       begin
56:       for i:=1 to n do
57:       Top:=Top-1;
58:       end;
59:     if (x<=Width div 2) then  {сдвигаемся по горизонтали}
60:       begin
61:       for i:=1 to n do
62:       Left:=Left+1;
63:       end
64:     else
65:       begin
66:       for i:=1 to n do
67:       Left:=Left-1;
68:       end;
69:     end;
70: end;
```

```
71:
72: end.
```

В процедуре NeinMove есть параметр

```
Shift: TShiftState
```

Он показывает, какие из управляющих клавиш (Alt, Shift, Ctrl) были нажаты. Но в данном обработчике нам эти параметры не нужны.

17.6. Установка пароля на программу

Во многих примерах, которые мы будем писать в дальнейшем, мы не будем изменять файл проекта. Поэтому у вас может появиться ощущение, что этот файл для оконных приложений священен, и прикасаться к нему нельзя. Чтобы у вас такого чувства не возникало, мы напишем программу, при входе в которую проверяется пароль (он вводится с помощью диалогового окна).

Пример 4: Защита программы паролем (приводится лишь dpr-файл, а файл с формой я оставил тот, который был по умолчанию).

```
1 : program Kennwort;
2 :
3 : uses
4 :   Forms,
5 :   Dialogs,
6 :   haupt in 'haupt.pas' {Form1};
7 :
8 : {$R *.res}
9 : var
10:   Mel:string; {Meldung - сообщение}
11: begin
12:   Mel:=InputBox('','Введите пароль','');
13:   if Mel<>'Anders' then
14:     begin
15:     ShowMessage('Неверный пароль');
16:     exit;
17:     end;
18:   Application.Initialize;
19:   Application.CreateForm(TForm1, Form1);
20:   Application.Run;
21: end.
```

17.7. Динамическое создание компонентов

Сейчас мы запрограммируем калькулятор, который сможет работать с числами в разных системах счисления. Для того чтобы обозначать цифры в системах, где основание больше десяти, мы будем использовать заглавные латинские буквы. Т.к. их только 26, то возможные основания ССч могут колебаться от 2 до 36. Для красоты мы сделаем так, чтобы на форме никогда не находились лишние клавиши, причем при уменьшении основания ССч будет увеличиваться размер клавиш (см. рис. 17.4).

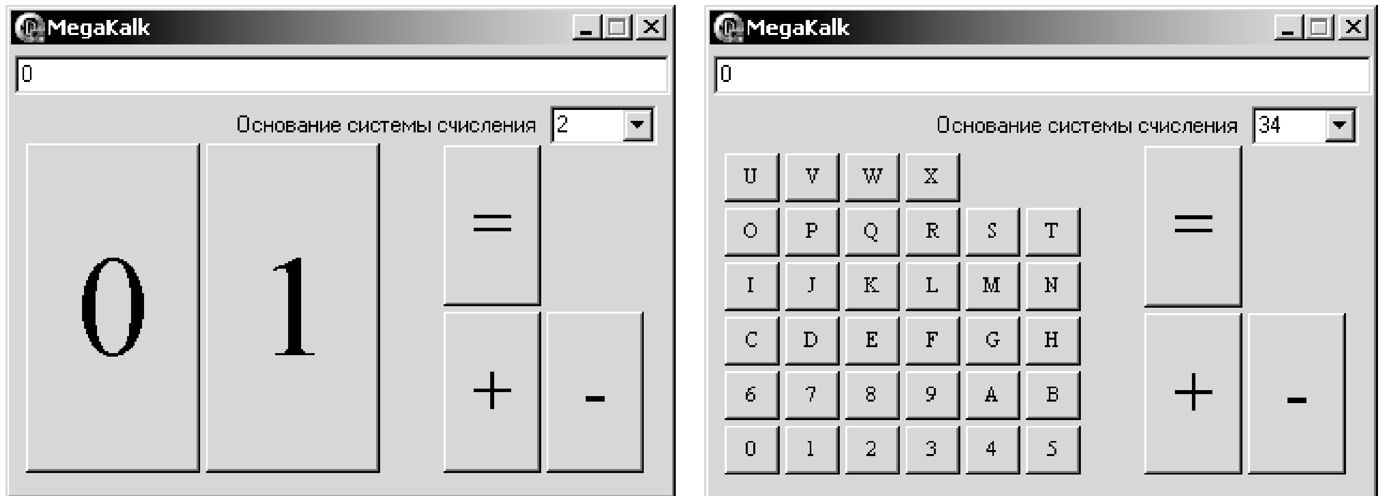


Рис 17.4 Форма для калькулятора

Из операций мы рассмотрим лишь сложение и вычитание. Другие операции вы легко сможете добавить сами. Для того чтобы написать эту программу, надо больше узнать о свойствах компонентов. Единственной алгоритмической трудностью является перевод чисел из одной ССч в другую – но эта задача для нас элементарна. Трудности иного плана возникнут при динамическом создании/удалении компонентов, - раньше мы весь интерфейс формы делали в Object Inspector, а теперь нам придется поработать вручную.

17.8. Общие свойства компонентов

- Компонентом называется любой класс-наследник класса TComponent.

Чтобы узнать всю информацию о свойствах компонентов, обращайтесь к справочной системе. Мы же остановимся на самом главном.

У каждого компонента есть следующие свойства:

`property Name: TComponentName;`	Имя компонента
`property Owner: TComponent;`	Владелец компонента
`property Components[Index: Integer]: TComponent;`	Список компонентов, владельцем которых является компонент
`property ComponentIndex: Integer;`	Индекс компонента в списке Components своего владельца
`property ComponentCount: Integer;`	Количество компонентов, которыми владеет данный компонент

Конструктор у класса TComponent имеет следующий вид:

```
constructor Create(AOwner: TComponent); virtual;
```

В конструктор надо передавать ссылку на его владельца. При этом одновременно изменяется свойство Owner компонента и ссылка на созданный компонент добавляется в экземпляр класса-владельца нового компонента.

Смысл «владения» в том, что когда вызывается деструктор какого-либо компонента, то одновременно вызываются деструкторы всех компонентов, владельцем которых он является.

Вообще говоря, компоненты не обязательно могут быть видимыми – например, диалоговые компоненты. Все компоненты, которые могут быть нарисованы, являются наследниками класса TControl, который, в свою очередь, является прямым наследником класса TComponent.

17.9. Свойства элементов управления (TControl)

Некоторые вы уже знаете – Left, Top, Height, Width.
Мы рассмотрим еще некоторые другие:

`property Visible: Boolean;`	True – если компонент видимый
`property Text: TCaption;`	Текстовое поле, связанное с компонентом
`property Caption: TCaption;`	Заголовок
`property Font: TFont;`	Шрифт, которым будет прорисовываться Caption
`property Parent: TWidgetControl;`	Родитель компонента
`property Color: TColor;`	Цвет заголовка

Родителями могут быть только объекты класса TWingetControl (это – наследник TControl). 2 важных дополнительных свойства этого класса приведены ниже:

`property ControlCount: Integer;`	Количество элементов массива Controls
`property Controls[Index: Integer]: TControl;`	Массив элементов, которыми управляет компонент

В конструкторе компонентов свойство Parent не устанавливается автоматически, поэтому если вы хотите назначить родителя компонента, то надо записать в свойство Parent ссылку на родителя.

17.10. События мыши и клавиатуры

Обработчик события – это свойство процедурного типа. Когда мы создавали обработчики события в Object-Inspector, автоматически создавалась процедура-обработчик и указатель на нее записывался в соответствующее свойство компонента.

Если мы хотим динамически создавать компоненты, то надо создавать обработчики событий вручную.

Давайте рассмотрим 5 основных обработчиков событий:

`property OnMouseDown: TMouseEvent;`	Обработчик нажатия кнопки мыши
`property OnClick: TNotifyEvent;`	Обработчик щелчка левой кнопкой мыши
`property OnMouseMove: TMouseMoveEvent;`	Перемещение указателя мыши по компоненту
`property OnMouseUp: TMouseEvent;`	Обработчик отпускания кнопки мыши
`property OnDblClick: TNotifyEvent;`	Обработчик двойного щелчка левой кнопкой мыши

```
type
     TMouseEvent = procedure (Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer) of object;
     TMouseMoveEvent = procedure(Sender: TObject; Shift:
TShiftState; X, Y: Integer) of object;
     TNotifyEvent = procedure (Sender: TObject) of object;
```

Sender содержит ссылку на компонент, над которым находился указатель мыши, X,Y – координаты указателя мыши в момент возникновения события.

Для определения, какая кнопка была нажата, есть спец. перечислимый тип TMouseButton.

```
type TMouseButton = (mbLeft, mbRight, mbMiddle);
```

TShiftState используется для получения специальной информации:

```
type TShiftState = set of (ssShift, ssAlt, ssCtrl, ssLeft, ssRight,
ssMiddle, ssDouble);
```

- ssShift — была нажата клавиша Shift
- ssAlt — была нажата клавиша Alt
- ssCtrl — была нажата клавиша Ctrl
- ssLeft — была нажата левая кнопка мыши
- ssRight — была нажата правая кнопка мыши
- ssMiddle — была нажата средняя кнопка мыши
- ssDouble — были нажаты одновременно левая и правая кнопки мыши

Чтобы написать обработчик события, надо написать процедуру соответствующего типа, и просто присвоить в соответствующее свойство адрес этой процедуры.

17.11. TComboBox

Для выбора основания ССч используется компонент TComboBox (находится во вкладке Standard).

У TComboBox есть свойство Items класса TStringList, в котором хранятся элементы, которые можно выбирать в комбосписке.

17.12. Описание работы калькулятора (Пример № 5)

Весь код я приводить здесь не буду. Мы рассмотрим лишь основные части проекта.

Подпрограммы, которые необходимы для написания калькулятора, можно разбить на 3 группы:

1. Подпрограммы, которые преобразуют числа из одной ССч в другую.
2. Создание кнопок и управление их размерами.
3. Обработчики события OnClick для всех кнопок.
4. Вспомогательные обработчики (FormCreate, FormGrundSelect).

Первую группу функций мы рассматривать не будем, - вы и сами разберетесь, что к чему.

Поля калькулятора

```
  private
    All:integer; //максимальное количество цифр
    KnNum:integer; //индекс первой кнопки
    OperQuant:integer;//количество кнопок со знаками и операциями
    OperInd:integer; //индекс, с которого начинаются операции
    Zahl1:integer;  //хранятся промежуточные числа
    WelcheZahl:integer; //какое слагаемое сейчас вводится
    R:TRect;  //прямоугольник, в котором будут расположены цифры
    R2:TRect;  //прямоугольник, в котором будут расположены знаки
операций
    VorherOper:integer; //индекс предыдущей операции, которая была
выполнена
// 0 - неарифметич. операция (нажатие цифры), 1 - сложение,
// 2 - вычитание,3 - после выдачи равенства
    LaufGrund:integer;  //текущее основание системы счисления
```

Теперь будут приведены основные процедуры.

2-я группа функций

```
// Создает кнопки и заполняет их начальными значениями (кроме
координат)
procedure SchaffeKnopfe;
//Прячет все кнопки с цифрами
procedure VerbergenKnopfe;
// Расставляет внутри прямоугольника R N кнопок, которые
// находятся в массиве Controls, начиная с номера ContrInd.
procedure Stellknopfe(R:TRect;N:integer;ContrInd:integer);

procedure TForm1.SchaffeKnopfe;
var
  k:integer;
  bt:TButton;
begin
 All:=36;  //сколько кнопок с цифрами
 KnNum:=self.ControlCount;//индекс первой кнопки совпадает с текущим
                          //количеством дочерних компонентов формы
 OperQuant:=3;// '+' '-' '='
 OperInd:=KnNum+All; // индекс первой Оперкнопки

//создаем кнопки и добавляем их в список Controls формы
 for k:=1 to All+OperQuant do
   begin
   bt:=TButton.Create(self);
   bt.Parent:=self;
   bt.Visible:=false; //изначально делаем все кнопки невидимыми
   end;

 for k:=KnNum to KnNum+All-1 do //заполняем значениями кнопки
```

```
   begin
     if k<10+KnNum then
       begin
       Controls[k].Name:='kn'+IntToStr(k-KnNum);
       TButton(Controls[k]).Caption:=IntToStr(k-KnNum);
       end
     else
       begin
       TButton(Controls[k]).Caption:=Chr(65+k-KnNum-10);
       Controls[k].Name:='kn'+Chr(65+k-KnNum-10);
       end;
//устанавливаем обработчик нажатия на Цифрокнопку
     TButton(self.Controls[k]).OnClick:=ZifferKlick;
   end;
//Изменяем параметры Оперкнопок
  Controls[operInd].Name:='knPl';
  TButton(Controls[operInd]).Caption:='+';
  TButton(Controls[operInd]).OnClick:=PlusKlick;

  Controls[operInd+1].Name:='knMin';
  TButton(Controls[operInd+1]).Caption:='-';
  TButton(Controls[operInd+1]).OnClick:=MinusKlick;

  Controls[operInd+2].Name:='knGl';
  TButton(Controls[operInd+2]).Caption:='=';
  TButton(Controls[operInd+2]).OnClick:=GleichKlick;
end;

procedure TForm1.Stellknopfe(R:TRect;N:integer;ContrInd:integer);
var
  i,j,k,a,b,aGut,bGut,diff:integer;
  Lan,Hohe:integer;
  ZwRaum:integer;
begin
 ZwRaum:=3; //расстояние между кнопками
//Ищем aGut, bGut - количество кнопок по вертикали и горизонтали
соответсвенно
//Эти числа выбираются так, чтобы разница между ними была
наименьшей.
 diff:=All;
 for b:=1 to All do
   for a:=b downto 1 do
     begin
     if ((b-a)>diff) or (b*a<n) then
       break;
     if b*(a-1)<n then
       begin
       if b-a<diff then
         begin
         aGut:=a;
         bGut:=b;
         diff:=b-a;
```

```
         end;
       break;
       end;
     end;
 Lan:=(R.Right-R.Left) div bGut;//длина кнопки (с межкнопочным
интерв.)
 Hohe:=(R.Bottom-R.Top) div aGut;//высота кнопки (с межкнопочным
интерв.)

 k:=ContrInd; //индекс первой кнопки
 for i:=1 to aGut do
   for j:=1 to bGut do
     begin
     Controls[k].Width:=Lan-ZwRaum;       // устанавливаем размеры
     Controls[k].Height:=Hohe-ZwRaum;
     Controls[k].Left:=R.Left+ZwRaum+(j-1)*Lan;
     Controls[k].Top:=R.Bottom-ZwRaum-i*Hohe;

     TButton(Controls[k]).Font.Name:='times New Roman';//назв.
шрифта
     TButton(Controls[k]).Font.Height:=Hohe div 2; //высота буквы
     self.Controls[k].Visible:=true;  //делаем кнопку видимой
     k:=k+1;
     if k=Contrind+n then  //если уже установили все нужные кнопки
       exit;
     end;
end;
```

3-я группа – обработчики нажатий на кнопки

Рассмотрим обработчики лишь для Цифрокнопки и Плюсокнопки.

Имя кнопки начинается с 2 букв – kn (der Knopf - кнопка), после которых следует цифра, которая стоит в заголовке кнопки.

Алгоритм обработчика ясен: если до этого было нажато равенство, то надо вводить новое число, поэтому старое число стирается из поля ввода и заменяется нажатой цифрой. Если предыдущая операция – не равенство, значит надо добавить новую цифру к числу, которое стоит в поле ввода. Кроме того, если в поле ввода стоит ноль, то нажатие новых нулей не изменяет поля ввода.

```
procedure TForm1.ZifferKlick(Sender:TObject);
begin
  if VorherOper=3 then //Если предыдущая операция - равенство
    begin
    EingFeld.Text:=TButton(Sender).Name[3]; //добавляем цифру
    vorheroper:=0;   //фиксируем, что была нажата клавиша
    end
  else
    if EingFeld.Text='0' then
      EingFeld.Text:=TButton(Sender).Name[3]
    else
```

```
      EingFeld.Text:=EingFeld.Text+TButton(Sender).Name[3];
//добавляем цифру
end;
```

При нажатии на кнопку плюс, если вводилось первое число, то теперь надо будет вводить второе, а сам текст сделать равным нулю. Если вводилось второе число, то отобразить результат сложения первого числа (которое хранится в переменной Zahl1) с числом, которое стоит в поле ввода.

```
procedure TForm1.PlusKlick(Sender:TObject);
begin
  case welcheZahl of
  1:
    begin
    Zahl1:=JedeStrZuInt(EingFeld.Text,LaufGrund);
    EingFeld.Text:='0';
    WelcheZahl:=2;
    end;
  2:
    begin
    if EingFeld.Text<>'0' then
      begin
      Zahl1:=Zahl1+JedeStrZuInt(EingFeld.Text,LaufGrund);
      EingFeld.Text:='0';
      end;
    end;
  end;
  VorherOper:=1;
end;
```

Вспомогательные функции

Так выглядит обработчик выбора нового основания ССч в Grund.

```
procedure TForm1.FormGrundSelect(Sender: TObject);
var
  etw:integer;
begin
  VerbergenKnopfe;
  etw:=JedeStrZuInt(EingFeld.Text,LaufGrund);
  LaufGrund:=StrToInt(Grund.Items[Grund.ItemIndex]);
  EingFeld.Text:=AusDezimZuJede(etw,LaufGrund);
  self.Stellknopfe(R,LaufGrund,KnNum);
end;
```

Начальные установки формы:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  i:integer;
begin
  WelcheZahl:=1;  //вводим сначала первое число
```

```
  VorherOper:=3;  //считаем, что изначально было равенство
  Zahl1:=0;

// Устанавливаем начальные координаты кнопок с цифрами
  R.Bottom:=height-30;
  R.Right:=width-160;
  R.Left:=5;
  R.Top:=55;
// Устанавливаем начальные координаты кнопок со знаками операций
  R2.Left:=R.Right+30;
  R2.Right:=width-20;
  R2.Top:=R.Top;
  R2.Bottom:=R.Bottom;
//создаем кнопки
  self.SchaffeKnopfe;

  self.Stellknopfe(R,{Grund.ItemIndex}8+2,KnNum);//расставляем цифры
  self.Stellknopfe(R2,OperQuant,OperInd);        //расставляем
операции

  EingFeld.Text:='0'; //заполняем текстовое поле
//заполняем ComboBox
  for i:=2 to All do
    Grund.Items.Add(IntToStr(i));
  Grund.ItemIndex:=8;

  LaufGrund:=Grund.ItemIndex+2;//Текущее основание ССч
end;
```

И, наконец, хотелось бы, чтобы нельзя было изменять размеры формы.

Для того чтобы нельзя было растягивать форму, надо изменить в Object Inspector свойство формы BorderStyle – установить значение bsSingle.

Чтобы нельзя было нажать клавишу «развернуть», можно зайти в разделе Properties во вкладку BorderIcons, и установить значение biMaximize равным False.

Вот мы и написали калькулятор!

17.13. Исключения

Исключительные ситуации (исключения) возникают, когда программа пытается выполнить некорректную операцию, например: деление на ноль, вычисление корня из отрицательных чисел (имеются в виду действительные и целые числа), попытка открыть несуществующий файл и т.п. В TP исключения нельзя было обрабатывать, что порождало ряд трудностей. Например, если программа не могла открыть несуществующий файл, то она прекращала свою работу и возвращала код ошибки. В Delphi обрабатывать исключительные ситуации можно, что мы и должны научиться делать.

Базовый класс для обработки исключений – класс Exception, являющийся прямым потомком класса TObject. Все классы-исключения являются наследниками класса Exception.

Для того чтобы обрабатывать исключение, в Delphi введен следующий оператор try:

1-я версия try:

```
try
  //операторы
except
  //обработчики исключений
else
 //операторы
end;
```

Часть программы, которая подозрительна на возникновение исключений, помещается после слова try. Если все операторы отработали нормально, то программа переходит к оператору следующему за оператором try. Если же возникло исключение, то просматриваются поочередно обработчики исключений, перечисленные в разделе except, пока не найдется такой обработчик, который сможет обработать исключение, которое возникло. Если ни один из перечисленных классов не способен обработать исключение, то выполняются операторы секции else.

2-я версия try:

```
try
  //операторы
finally
  //операторы
end;
```

Если в промежутке от try до finally возникло исключение, то все последующие операторы пропускаются и управление передается в секцию finally. Если же все операторы были успешно выполнены, то управление все равно передается операторам в секции finally.

Кстати, внутри обработчиков исключений нельзя использовать оператор goto.

Пример 6: Использование исключений.

Создайте форму с 2-мя текстовыми полями (типа TEdit) и одной кнопкой. По нажатию кнопки во второе поле должно записываться значение квадратного корня из числа, записанного в первом поле.

При таких вычислениях может возникнуть 2 исключения: если строка, написанная в первом поле, вообще не является числом (класс EConvertError), либо было введено отрицательное число, следовательно, операция вычисления квадратного корня недопустима и возникнет исключение EInvalidOp.

Обработчик нажатия на кнопку будет иметь следующий вид:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  x:real;
begin
  try
    x:=StrToFloat(edit1.text); //Может возникнуть EConvertError
    edit2.text:=FloatToStr(sqrt(x)); //Может возникнуть EInvalidOp
  except
```

```
    on EConvertError do
      showMessage('неверно введено число');
    on EInvalidOp do
      showMessage('отрицательное число');
  end;
end;
```

Как видите, обработка исключения может быть не только суровой необходимостью, но и благом: мы могли бы обойтись и без использования исключений, но для этого надо было бы самостоятельно проверить правильность ввода строки – т.е. проверить, было ли действительно введено действительное число, а для этого нам пришлось бы фактически писать собственную процедуру перевода строки в число, которая при неверно сформированной строке выполняла необходимые нам действия, а это значительно дольше, чем обработать исключение EConvertError.

Можно создавать и собственные классы исключений. Зачем они нужны, если ОС все равно сама не сможет генерировать исключения таких классов? Лишь затем, чтобы самостоятельно их вызывать и затем обрабатывать.

В следующем примере мы создадим класс, который будет представлять собой «защищенный» массив. В обычных массивах можно обращаться к элементам, индексы которых не входят в диапазон, который объявлен для данного массива. При этом могут портиться данные, которые находятся в этих байтах. Мы с вами создадим класс, у которого есть свойство-массив, причем при выходе за границы массива или при попытке присвоить элементу массива отрицательное число будут генерироваться исключения.

Пример 7: Обработка пользовательских исключений.

```
1 : program RaiseExc;
2 :
3 : {$APPTYPE CONSOLE}
4 :
5 : uses
6 :   SysUtils,
7 :   Dialogs;
8 :
9 : const
10:   n=5;
11:
12: type
13:   ESchlechteIndex = class(Exception);//выход за границы массива
14:   ENegativeZahl = class(Exception); //отрицательное число
15:
16:   Mas = array [1..n] of integer;
17:
18:   MasKlass = class
19:   private
20:     R:Mas;
21:     procedure StellZahl(Index:integer;G:integer);
22:     function GibZahl(Index:integer):integer;
23:   public
```

```
24:     property Mass[Ind:integer]:integer read GibZahl write
StellZahl;
25:     function ZurZeile:string;  //преобразует в MasKLass строку
26:   end;
27:
28: procedure MasKlass.StellZahl(Index:integer;G:integer);
29: begin
30:   if (index>n) or (index<1) then //если индекс вышел за границы
массива
31:     raise ESchlechteIndex.Create('Индекс вышел за границу
массива');
32:   if G>=0 then
33:     R[Index]:=G
34:   else
35:     raise ENegativeZahl.Create('В MasKlass не могут быть
отрицательные числа');
36: end;
37:
38: function MasKlass.GibZahl(Index:integer):integer;
39: begin
40:   if (index>n) or (index<1) then //если индекс вышел за границы
массива
41:     raise ESchlechteIndex.Create('Индекс вышел за границу
массива')
42:   else
43:     result:=R[index];
44: end;
45:
46: function MasKlass.ZurZeile:string;
47: var
48:   i:integer;
49: begin
50:   Result:='';
51:   for i:=1 to n do
52:     Result:=Result+IntToStr(R[i])+' ';
53: end;
54:
55: var
56:   MK:MasKlass;
57:   i:integer;
58: begin
59:   MK:=MasKlass.Create;
60:   for i:=1 to n+3 do
61:     try
62:       MK.Mass[i]:=i - 2;
63:     except
64:       on ESchlechteIndex do  //если вышли за границы массива
65:         ShowMessage('Плохой индекс');
66:       on ENegativeZahl do //если в массив пытаются записать
отрицательное число.
67:         MK.Mass[i]:=0;
68:     end;
```

```
69:   writeln(MK.ZurZeile);
70:   readln;
71: end.
```

17.14. Класс TList

Класс TList и его наследники часто используются в компонентах, поэтому мы рассмотрим его с вами. TList – это класс, который моделирует список с помощью динамического массива нетипизированных указателей. Это означает, что хотя по своей сути это – массив, но методы класса реализованы так, как будто это на самом деле список.

Вспомним особенности использования динамических массивов в памяти. При добавлении нового элемента к массиву может случиться, что непосредственно за массивом в памяти нет свободного места для размещения нового элемента. Тогда надо будет найти в динамической памяти область, достаточного размера, чтобы она могла вместить массив вместе с новым элементом, затем скопировать в эту область весь массив из предыдущего места хранения, добавить к нему новый элемент, а старое место пребывания вернуть в кучу. Как вы понимаете, такая процедура занимает достаточно много времени, поэтому класс TList кроме базовых возможностей работы с динамическим массивом обладает рядом функций, позволяющих лучше использовать память.

Во-первых, класс TList всегда занимает в куче места не меньше, чем его количество элементов. Ключевое слово в предыдущей фразе – «не меньше», т.к. можно зарезервировать память «про запас».

Общее количество элементов, занятых массивом (с теми, которые запасены впрок), находится в свойстве Capacity. Количество занятых элементов находится в свойстве Count.

Получить доступ к любому элементу класса можно с помощью свойства

```
property Items[Index: Integer]: Pointer;
```

Основные методы:

procedure Clear; virtual;

Удаляет все элементы в списке.

procedure Delete(Index: Integer);

Удаляет элемент в списке с заданным индексом. При этом сама память, связанная с элементом не освобождается (за это отвечает свойство capacity).

function Add(Item: Pointer): Integer;

Добавляет элемент Item в конец списка и возвращает его индекс.

procedure Insert(Index: Integer; Item: Pointer);

Вставляет элемент Item на место Index (при этом все элементы массива, начиная с места Index сдвигаются на 1 вправо).

procedure Sort(Compare: TListSortCompare);
type TListSortCompare = function (Item1, Item2: Pointer): Integer;

Сортирует массив с помощью функции типа TListSortCompare. Функция Compare должна возвращать:

положительное число, если Item1>Item2

0, если Item1=Item2

отрицательное число, если Item1<Item2

Пример 8: Использование TList.

Смысл программы в том, что создается массив записей, каждая из которых представляет собой дробь. Процедура
procedure FulleLst(var T:TList;n:integer);
заполняет список n случайными значениями. Заметьте, что перед заполнением списка устанавливается значение свойства capacity (строка 39). Это надо как раз для того, чтобы не приходилось много раз расширять массив.
Для сортировки списка будет использоваться (строка 75) функция
`function Vergleiche(p1,p2:pointer):integer;`

```
1 : unit Haupt;
2 :
3 : interface
4 :
5 : uses
6 :   Windows, Messages, SysUtils, Variants, Classes, Graphics,
7 :   Controls, Forms, Dialogs, StdCtrls;
8 :
9 : type
10:   TForm1 = class(TForm)
11:     Memo1: TMemo;
12:     Button1: TButton;
13:     procedure Button1Click(Sender: TObject);
14:   private
15:     { Private declarations }
16:   public
17:     { Public declarations }
18:   end;
19:
20: type
21:   Rat = record
22:     x,y:integer;
23:   end;
24:
25:   pRat=^Rat;
26:
27: var
28:   Form1: TForm1;
29:   Lst:TList;
30: implementation
31:
32: {$R *.dfm}
33: procedure FulleLst(var T:TList;n:integer);
34: var
35:   R:pRat;
36:   i:integer;
37: begin
38:   T:=TList.Create;  //создаем список
39:   T.Capacity:=n;  //устанавливаем его длину
40:   randomize;
```

```
41:   for i:=1 to n do
42:     begin
43:     new(R);
44:     r^.x:=random(n);
45:     repeat         //знаменатель не должен быть равен 0
46:       r^.y:=random(n);
47:     until r^.y<>0;
48:     T.Add(R);  //добавляем элемент в список
49:     end;
50: end;
51:
52: //Функция сравнения для элементов списка
53: function Vergleiche(p1,p2:pointer):integer;
54: begin
55:   if pRat(p1).x*pRat(p2).y>pRat(p1).y*pRat(p2).x then
56:     result:=1
57:   else
58:     if pRat(p1).x*pRat(p2).y=pRat(p1).y*pRat(p2).x then
59:       result:=0
60:     else
61:       result:=-1;
62: end;
63:
64: procedure TForm1.Button1Click(Sender: TObject);
65: var
66:   i:integer;
67: begin
68:   FulleLst(Lst,100);
69:   Memo1.Lines.Add(IntToStr(Lst.Count));
70:
71:   for i:=1 to Lst.Count do
72:     Memo1.Lines.Add(IntToStr(i)+': '+IntToStr(pRat(Lst.Items[i-
1])^.x)+ '/'+ IntToStr(pRat(Lst.Items[i-1])^.y));
73:
74:   Memo1.Lines.Add('Отсортированный массив');
75:   Lst.Sort(Vergleiche); //Сортируем список с помощью функции
Vergleiche
76:   for i:=1 to Lst.Count do
77:     Memo1.Lines.Add(IntToStr(i)+': '+IntToStr(pRat(Lst.Items[i-
1])^.x)+ '/' +IntToStr(pRat(Lst.Items[i-1])^.y));
78: end;
79: end.
```

17.15. Процессы и потоки

Напомню основные определения:

- Процесс – выполняемая программа, включая текущие значения счетчика команд, регистров переменных, используемые ресурсы (например, открытые файлы).
- Поток – это выполняемая программа (набор команд), включая регистры и переменные стека.

Поток может находиться в следующих состояниях:

1. Существующий
2. Несуществующий
3. Работающий
4. Приостановленный (т.е. он существует, но в данный момент не выполняется).

Для работы с потоками в Delphi есть специальный класс TThread. В этом классе есть свойство

```
property Suspended: Boolean;
```

которое показывает, приостановлен ли поток. Если Suspended = true, то поток приостановлен. В противном случае он – работающий.

С этим свойством можно работать напрямую, устанавливая ему нужное значение, а можно использовать и следующие процедуры:

```
procedure Resume; {переводит поток из состояния «приостановленный» в состояние «работающий»}
procedure Suspend; {приостанавливает работу потока}
```

Чтобы продемонстрировать полезность потоков, в примере мы будем сортировать большой массив чисел методом пузырька (конечно, большие массивы пузырем никто не сортирует, но нам сейчас главное – чтобы процедура работала долго). Если не выделить для сортировки отдельный поток, то пользователь может надолго утратить контроль за окном. А выделение дополнительного потока эту проблему легко снимет.

Основной модуль HauptFlut содержит 2 процедуры работы с массивами – случайного заполнения и сортировки. При щелчке на кнопку Button2 будет производиться сортировка массива без создания нового потока. При щелчке на кнопку Button1 будет создаваться дополнительный поток.

Класс потока можно создать, выбрав при создании нового файла ThreadObject (естественно, можно это сделать просто написав нужный класс).

Класс потока написан в модуле FlutSort. Любой поток должен быть наследником класса TThread. Основной подпрограммой потока является процедура Execute. В самом TThread эта процедура объявлена как абстрактная и, следовательно, должна переопределяться во всех потомках этого класса.

Т.к. все потоки находятся в одном и том же адресном пространстве, то надо каким-то образом синхронизировать работу потоков с глобальными переменными.

Для синхронизации работы потоков в классе TThread служит процедура Synchronize, единственным параметром которого является процедура без параметров.

Вспомогательный поток (объект класса Sorting) в процедуре Execute сортирует глобальный массив и потом записывает отсортированный массив в Memo1. Для того, чтобы дописывать строки используется процедура Schr, которая является параметром процедуры Synchronize.

Пример 9: Сортировка в дополнительном потоке.

```
1 : unit Hauptflut;
2 :
3 : interface
4 :
5 : uses
6 :   Windows, Messages, SysUtils, Variants, Classes, Graphics,
7 :   Controls, Forms,Dialogs, StdCtrls,
```

```
8 :   FlutSort; //модуль, в котором хранится класс-наследник TThread
9 :
10: type
11:   TForm1 = class(TForm)
12:     Memo1: TMemo;
13:     Button1: TButton;
14:     Memo2: TMemo;
15:     Button2: TButton;
16:     procedure FormCreate(Sender: TObject);
17:     procedure Button1Click(Sender: TObject);
18:     procedure Button2Click(Sender: TObject);
19:   private
20:     { Private declarations }
21:   public
22:     { Public declarations }
23:   end;
24:
25: const
26:   n=40000;
27: type
28:   Mas=array [1..n] of integer;
29: var
30:   Form1: TForm1;
31:   M:Mas;
32:   Sort:Sorting; //поток
33:
34: implementation
35: {$R *.dfm}
36:
37: procedure RandMas(var M:Mas);
38: var
39:   i:integer;
40: begin
41:   randomize;
42:   for i:=1 to n do
43:     M[i]:=random(1000000);
44: end;
45:
46: procedure BlaseSort(var M:Mas);
47: var
48:   i,j,tmp:integer;
49: begin
50:   for i:=n downto 1 do
51:     for j:=1 to i-1 do
52:       if M[j]<M[j+1] then
53:         begin
54:         tmp:=M[j];
55:         M[j]:=M[j+1];
56:         M[j+1]:=tmp;
57:         end;
58: end;
59:
```

```
60: procedure TForm1.FormCreate(Sender: TObject);
61: begin
62:   Memo1.Left:=0;
63:   Memo1.Top:=0;
64:   Memo1.Width:=Form1.Width div 2;
65:   Memo1.Height:=Form1.Height div 2;
66:   RandMas(M);
67:   Sort:=Sorting.Create(true); //Создаем объект Sort
68: end;
69:
70: procedure TForm1.Button1Click(Sender: TObject);
71: begin
72:   try  //поток Sort может не существовать
73:     if Sort.Suspended=false then  //Если поток работает
74:       Sort.Suspend  {Останавливаем поток}
75:     else       //если сейчас остановлен (но существует !)
76:       Sort.Resume;  {Возобновляем работу потока}
77:   except     //если не существует
78:     on EThread do
79: //      ShowMessage('Поток уже завершил работу')
80:   end;
81:
82: end;
83:
84: //просто сортируем массив, не выделяя дополнительного потока
85: procedure TForm1.Button2Click(Sender: TObject);
86: begin
87:   BlaseSort(M);
88: end;
89:
90: end.
------------------------- Класс потока ----------------------------
1 : unit FlutSort;
2 :
3 : interface
4 :
5 : uses
6 :   Classes,SysUtils;
7 :
8 : type
9 :   Sorting = class(TThread)
10:   private
11:     { Private declarations }
12:   protected
13:     s:string;   //строка, которая должна дописываться в Memo.
14:     procedure Execute; override;
15:     procedure Schr;
16:   end;
17:
18: implementation
19:
20: uses
```

```
21:   HauptFlut; //основной модуль (присоединить в интерфейсной
части нельзя)
22: {
23: Schr - процедура, с помощью которой проводится синхронизация
24: с основным потоком процесса.
25: }
26: procedure Sorting.Schr;
27: begin
28:   Form1.Memo1.Lines.Add(S);
29: end;
30:
31: //Execute - основная процедура потока.
32: // Когда она заканчивает свою работу, поток исчезает.
33: procedure Sorting.Execute;
34: var
35:   i,j,tmp:integer;
36: begin
37:   for i:=n downto 1 do
38:     for j:=1 to i-1 do
39:       if M[j]<M[j+1] then
40:         begin
41:         tmp:=M[j];
42:         M[j]:=M[j+1];
43:         M[j+1]:=tmp;
44:         end;
45:   s:='';
46:   i:=1;
47:   while i<=n do
48:     begin
49:     for j:=1 to 5 do //формируем строку
50:       begin
51:       if (i>n) then
52:         break;
53:       s:=s+IntToStr(M[i])+' ';
54:       inc(i);
55:       end;
56:     Synchronize(Schr); {дописываем строку в поле}
57:     s:='';
58:     end;
59: end;
60: end.
```

Задачи

1. Изменить программу с убегающей кнопкой так, чтобы она не могла выходить за границы формы.
2. Модернизировать калькулятор так, чтобы он мог работать с вещественными числами (в любой ССч). Также добавьте дополнительные операции: деление, возведение в степень а также вычисление стандартных функций: синуса, косинуса и т.д.

Глава 18: Графика

С точки зрения пользователя, рисовать можно прямо на форме, за исключением компонентов, которые на ней расставлены. Но для того, чтобы не создавать путаницы, графическими возможностями заведует класс TCanvas. У формы есть объект этого класса. В состав класса TCanvas входят, помимо всего прочего, функции рисования простых геометрических фигур и три свойства: Pen типа TPen (перо), Brush типа TBrush (кисть), Font типа TFont (шрифт).

Перо отвечает за тип и цвет выводимых на канву линий, кисть – за заливку сплошных областей, а шрифт – за тип и размер шрифта, которым будет выводиться на канву текст.

18.1. Цвет кисти и пера

Любой цвет можно задать в виде наложения синего, красного и зеленого цветов. Задавая разные интенсивности этих цветов, мы будем получать различные результирующие цвета. В Delphi цвет определяется так (его объявление находится в модуле Graphic):

```
type
  TColor = -$7FFFFFFF-1..$7FFFFFFF;
```

Т.е. цвет задается 4-байтовым целым числом.

Старший байт – это номер палитры. Чтобы использовать стандартную палитру, надо устанавливать значение старшего байта равным 00.

Следующие 3 байта задают интенсивность синего, зеленого и красного цвета соответственно. Например:

$00FF0000 – насыщенный синий цвет

$0000FF00 – насыщенный зеленый цвет

$000000FF – насыщенный красный цвет

Многие распространенные цвета можно задавать также с помощью встроенных констант, например:

clWhite, clRed – константы, задающие белый и красный цвета соответственно.

Другие константы вы можете посмотреть в справочной системе (наберите TColor type)

У классов TBrush, TPen есть свойство, отвечающее за цвет заливки областей и цвета выводимых линий соответственно:

```
property Color:TColor;
```

18.2. Рисование многоугольников

Следующая процедура рисует прямоугольник текущим пером, и закрашивает его текущей кистью.

```
procedure Rectangle(X1, Y1, X2, Y2: Integer); overload;
procedure Rectangle(const Rect: TRect); overload;
```

Тип TRect определяется так:

```
type
  TPoint = packed record
  X: Longint;
  Y: Longint;
end;

  TRect = packed record
  case Integer of
    0: (Left, Top, Right, Bottom: Integer);
    1: (TopLeft, BottomRight: TPoint);
end;
```

Ключевое слово packed влияет на способ хранения структурированных типов (записей, массивов, классов и т.д.). Можете набрать слово packed в справочной системе, чтобы получить больше информации.

Чтобы нарисовать незакрашенный прямоугольник, можно использовать процедуру

```
procedure FrameRect(const Rect: TRect);
```

Но имейте в виду, что граница прямоугольника рисуется в этой процедуре цветом кисти.

Чтобы рисовать пером, надо использовать процедуру Polygon, которая рисует многоугольник по заданному массиву вершин (ребро – это линия между соседними вершинами).

```
procedure Polygon(Points: array of TPoint);
```

18.3. Рисование линий

```
procedure MoveTo(X, Y: Integer);
```

Устанавливает новую позицию пера.

```
procedure LineTo(X, Y: Integer);
```

Рисует линию от текущего положения пера до точки (x, y). При этом новая позиция пера становится равной (x, y).

На самом деле в LineTo рисуется не линия, а отрезок, начало которого хранится в свойстве PenPos (позиция пера) канвы.

Кроме линий и прямоугольников в классе TCanvas есть еще довольно много графических примитивов. Вы можете прочитать о них, посмотрев методы класса TCanvas в справочной системе.

Пример 1: Построение правильного многоугольника.

Создайте новый проект, и на форму поместите простую кнопку (Button1). По нажатию этой кнопки на канве будет рисоваться правильный многоугольник.

```
procedure
VielEck(n:integer;x,y:integer;a:integer;fi0:real;C:TCanvas);
var
  i:integer;
  x1,x2,y1,y2:integer;
  fi:real;  //угол, на который надо повернуть вершину многоуг.
begin
  x1:=x+round(a*cos(fi0));
```

```
  y1:=y+round(a*sin(fi0));
  fi:=fi0+2*Pi/n;
  C.MoveTo(x1,y1); //переходим в точку, с которой будут чертиться
линии
  for i:=0 to n-1 do //чертим все линии
    begin
    x2:=x+round(a*cos(fi)); //координаты следующей вершины
    y2:=y+round(a*sin(fi));
    C.LineTo(x2,y2); //чертит линию от текущего положения до точки
(x2,y2)
    fi:=fi+2*Pi/n;
    end;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  with Canvas do
    begin
    Pen.Color:=clWhite;//устанавливаем белый цвет пера
    Brush.Color:=clWhite;//устанавливаем белый цвет кисти
    Rectangle(clipRect);//рисуем белый прямоугольник на весь экран
    Pen.Color:=clRed;  //устанавливаем красный цвет пера
    VielEck(4,300,300,150,0,Form1.Canvas);
    end;
end;
```

По-немецки многоугольник – das Polygon, но мне кажется, что гораздо красивее было бы назвать его das Vieleck (по аналогии с Viereck – четырехугольник; само слово viel означает много). Поэтому я назвал процедуру VielEck. Ее параметры – количество углов, центр окружности, описанной около многоугольника, радиус описанной окружности и угол относительно положительного направления оси OX, на который надо повернуть многоугольник и канва, на которой сам Vieleck будет рисоваться.

Сначала рассмотрим обработчик нажатия кнопки. Работа ведется со свойством Canvas (которое принадлежит классу TForm).

Чтобы очистить экран закрашиваем весь фон белым цветом.

ClipRect, который передается в процедуру рисования прямоугольника – это специальное свойство канвы:

`property ClipRect:TRect;`	Содержит прямоугольник, который должен быть перерисован.

Когда форма только создается, ClipRect содержит размеры всей канвы.

Если вы еще не запустили программу, то сделайте это. Вы увидите, что кнопка находится поверх канвы, и на ней ничего не рисуется. Теперь измените размеры экрана, перетащите окно так, чтобы оно одним краем вылезло за границы экрана и верните в исходное положение или сверните окно, а затем восстановите его. Вы увидите, что изображение на канве не перерисовывается само по себе, в отличие от кнопок и других компонентов.

18.4. Рисуем шашечную доску

У формы есть специальное событие OnPaint, которое возникает тогда, когда надо перерисовать форму. Если написать обработчик этого события, то мы сможем выполнять специальные действия при перерисовке изображения канвы.

При изменении размеров формы возникает событие OnResize. В следующем примере мы будем рисовать шашечную доску (иногда ее называют шахматной, но исторически раньше появились шашки, поэтому правильнее писать «шашечная доска»). Мы сделаем так, чтобы она перерисовывалась, когда это необходимо и, кроме того, чтобы ее размеры менялись в зависимости от размеров формы.

Пример 2: Шашечная доска.

```
1 : unit BrettU;
2 :
3 : interface
4 :
5 : uses Graphics,Types,Forms,Math;
6 :
7 : type
8 :   Brett = class {das Brett - доска}
9 :   private
10:       Langex:byte;{количество клеток по горизонтали}
11:       Langey:byte;{количество клеток по вертикали}
12:       Seite:byte;{длина стороны клетки (в пикселях)}
13:       x,y:integer;{координаты верхнего левого угла доски}
14:       farbe1:TColor;{цвет "белых" клеток}
15:       farbe2:TColor;{цвет "черных" клеток}
16:       ErwSeite:integer; //желаемая ширина клетки
17:   public
18:       procedure FestSt(aLangex,aLangey,aSeite:byte;
ax,ay:integer; afarbe1,afarbe2:TColor);
19:       constructor Bilde(aLangex,aLangey,aSeite:byte;
ax,ay:integer; afarbe1,afarbe2:TColor);
20:       procedure Male(C:TCanvas);
21:       procedure StellFormat(Frm:TForm);
22:   end;
23:
24: implementation
25:
26: constructor Brett.Bilde(aLangex,aLangey,aSeite:byte;
ax,ay:integer; afarbe1,afarbe2:TColor);
27: begin
28:   FestSt(aLangex,aLangey,aSeite,ax,ay,afarbe1,afarbe2);
29: end;
30:
31: procedure Brett.FestSt(aLangex,aLangey,aSeite:byte;
ax,ay:integer; afarbe1,afarbe2:TColor);
32: begin
33:       Langex:=aLangex;
34:       Langey:=aLangey;
35:       Seite:=aSeite;
```

```
36:       x:=ax;
37:       y:=ay;
38:       farbe1:=afarbe1;
39:       farbe2:=afarbe2;
40:       ErwSeite:=Seite;
41: end;
42:
43: procedure Brett.StellFormat(Frm:TForm);
44: var
45:   EinzugBr,EinzugH:integer;{отступ от краев (по ширине и
высоте)}
46:   R:TRect;
47: begin
48:   EinzugBr:=15;
49:   EinZugH:=15;
50:   {Вычисляем текущие координаты клиентской части формы}
51:   R.Left:=Frm.Left;
52:   R.Top:=Frm.Top;
53:   R.Right:=Frm.Left+Frm.ClientWidth;
54:   R.Bottom:=Frm.Top+Frm.clientHeight;
55:
56:   {Длина и ширина поля}
57:   Seite:=min(min(ErwSeite,(R.Right-R.Left-2*EinzugBr) div
LangeX), (R.Bottom-R.Top-2*EinzugH) div LangeY);
58:
59:   x:=(R.Right-R.Left - Seite*LangeX) div 2;
60:   y:=(R.Bottom-R.Top - Seite*LangeY) div 2;
61:
62: end;
63:
64: procedure Brett.Male(C:TCanvas);
65: var
66:   Pfrb,Bfrb:TColor;
67:   i,j:byte;
68:   R:TRect;
69: begin
70:   Pfrb:=C.Pen.Color;
71:   Bfrb:=C.Brush.Color;
72:   C.Brush.Color:=farbe1;
73:   C.Pen.Color:=farbe2;
74: {заполняем всю доску "белым" цветом}
75:   R.Left:=x;
76:   R.Right:=x+Langex*Seite;
77:   R.Top:=y;
78:   R.Bottom:=y+Langey*Seite;
79:   C.Rectangle(R);
80:
81:   C.Brush.Color:=farbe2;
82: {Заполняем нужные клетки "черным" цветом}
83:     for i:=1 to (Langex div 2) do
84:       for j:=1 to (Langey div 2) do
```

```
85:          C.Rectangle(x+(2*i-1)*Seite,y+(2*j-
2)*Seite,x+2*i*Seite,y+(2*j-1) *Seite);
86:
87:      for i:=1 to (Langex div 2) do
88:        for j:=1 to (Langey div 2) do
89:          C.Rectangle(x+(2*i-2)*Seite,y+(2*j-1)*Seite,x+(2*i-
1)*Seite, y+(2*j)*Seite);
90:
91:    C.FrameRect(R);
92:    C.Pen.Color:=Pfrb;
93:    C.Brush.Color:=Bfrb;
94: end;
95: END.
```

В процедуре Male используется FrameRect (см. «рисование прямоугольников»). Сам алгоритм рисования можете разобрать самостоятельно. StellFormat («установи формат») устанавливает размеры доски относительно формы. У доски есть желаемая ширина клетки. Если доска с такой шириной клетки может поместиться на канву, то она рисуется именно с таким размером по центру канвы. Если вся доска не влезает на канву, то размер выбирается так, чтобы она влезла и еще оставалось место до края канвы (EinzugBr – отступ по горизонтали, EinzugH – отступ по вертикали). Теперь рассмотрим основной модуль.

```
1 : unit haupt;
2 :
3 : interface
4 :
5 : uses
6 :   Windows, Messages, SysUtils, Variants, Classes, Graphics,
7 :   Controls, Forms,Dialogs,BrettU;
8 :
9 : type
10:   TForm1 = class(TForm)
11:     procedure FormCreate(Sender: TObject);
12:     procedure FormPaint(Sender: TObject);
13:     procedure FormResize(Sender: TObject);
14:   private
15:     B:Brett;
16:   public
17:     { Public declarations }
18:   end;
19:
20: var
21:   Form1: TForm1;
22:
23: implementation
24:
25: {$R *.dfm}
26:
27: procedure TForm1.FormCreate(Sender: TObject);
28: begin
```

```
29:   B:=Brett.Bilde(40,30,15,100,100,clYellow,clBlack);
30: end;
31:
32: procedure TForm1.FormPaint(Sender: TObject);
33: begin
34:   B.Male(Canvas);
35: end;
36:
37: procedure TForm1.FormResize(Sender: TObject);
38: begin
39:   with Canvas do
40:     begin
41:     Pen.Color:=Brush.Color;
42:     Rectangle(ClipRect);
43:     end;
44:   B.StellFormat(Form1);
45:   B.Male(Canvas);
46: end;
47: end.
```

Сами видите, что я добавил к форме одно закрытое поле – доску.

При создании формы возникает событие OnCreate, которое обрабатывается методом FormCreate, в которой создается объект B. Сразу после этого форма должна быть перерисована, поэтому возникает событие OnPaint, которое обрабатывается с помощью процедуры FormPaint. FormResize, обработчик события OnResize, вызывается после изменения размеров формы. В нем закрашивается все содержимое формы, устанавливаются новые параметры доски и рисуется ее обновленный вариант.

18.5. Фракталы

Термин «фрактал» ввел Бенуа Мандельброт. Но первые фрактальные множества появились задолго до работ Мандельброта. На рисунке 18.1 вы видите множество Кантора, которое строится на базе простого отрезка (0-я итерация). На первом шаге из отрезка выбрасывается средняя треть. Затем из двух оставшихся третей также выбрасываются центральные трети и т.д. до бесконечности. То, что осталось от отрезка после бесконечного количества выбрасываний и есть множество Кантора (K).

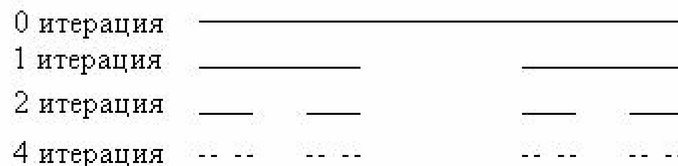


Рис. 18.1. Множество Кантора (0, 1, 2, 4-я итерации)

Что в нем такого необычного, спросите вы? Но оказывается, что несмотря на то, что суммарная длина выброшенных отрезков равна длине исходного отрезка, количество точек множества Кантора (т.е. его мощность) равно количеству точек исходного отрезка (это означает, что можно найти функцию, которая каждой точке K поставит в соответствие некоторую точку отрезка, причем разным точкам K

соответствуют различные точки отрезка и для любой точки x отрезка существует точка из K, которая отображается в точку x)!

Другим интересным примером является кривая Коха[17], с которой вы познакомились в начале главы «Рекурсия».

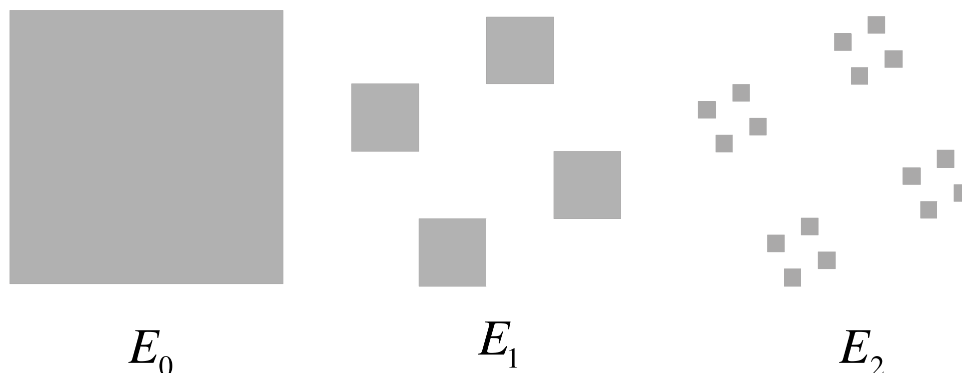


Рис. 18.2. Пыль Кантора (0-я, 1-я и 2-я итерации)

Еще одним монстром является кривая Пеано, которая полностью заполняет собой квадрат (т.е. проходит через каждую его точку). На рисунке вы видите первые 3 итерации построения этой кривой. Сама же кривая получится после бесконечного числа итераций.

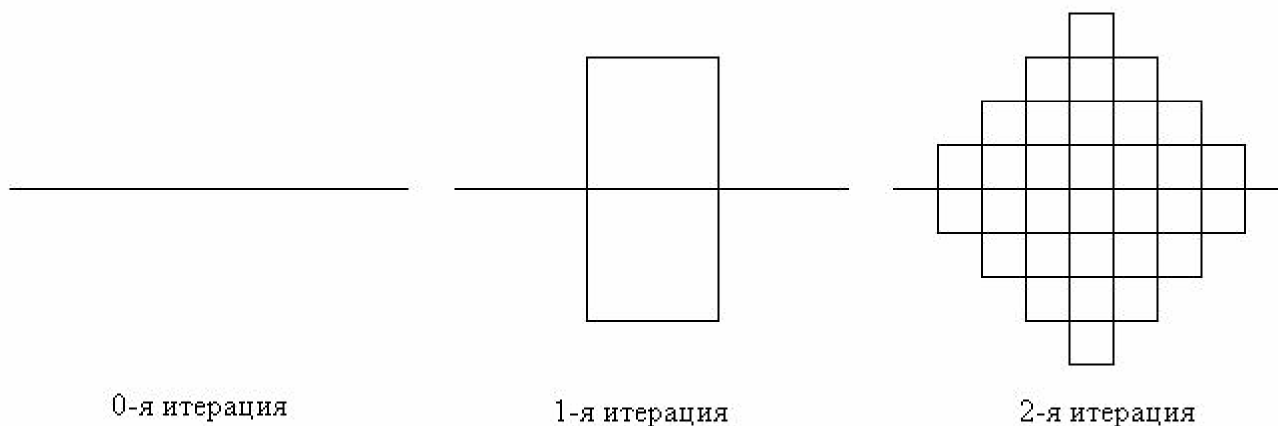


Рис. 18.3. Кривая Пеано

Обычно под фракталом понимают множество дробной размерности. Но для того, чтобы это понятие обрело смысл, надо определить, что такое размерность и каким образом она умудряется быть дробной.

Начнем мы с простейшего объекта – квадрата. Пусть на дан квадрат Q со стороной 3 см. Какое минимальное количество (N) квадратов со стороной δ см надо, чтобы покрыть квадрат Q? Пусть $\delta = 1$. Тогда $N = 9$.

$$\delta = \frac{1}{2} \Rightarrow N = 9 \cdot 4 = 9\delta^{-2}$$

---

[17] В различных книгах имя этого ученого переводилось так: Гельг фон Кох, Хельге фон Кох и даже Хельга фон Кох (Хельга – женское имя).

$\delta = \frac{1}{2^k} \Rightarrow N = 9 \cdot 4^k = 9\delta^{-2}$

Значит можно найти сколь угодно малое число δ такое, что минимальное количество квадратов со стороной δ, которыми можно покрыть исходный квадрат Q, будет равняться $9\delta^{-2}$ (9 – площадь квадрата, 2 – размерность плоскости).

Давайте теперь покрывать куб со стороной 3 см кубиками со стороной δ. По аналогии с предыдущим случаем получим:

$\delta = \frac{1}{2^k} \Rightarrow N = 27 \cdot 8^k = 27\delta^{-3}$ (27 – объем куба, а 3 – размерность пространства)

Назовем несущим пространством пространство, элементами которого мы покрываем множество, размерность которого мы хотим выяснить.

В двух разобранных примерах мы проверили, что покрытии множества некоторыми квадратами несущего пространства кол-во необходимых квадратов N будет равно

$$N = c\delta^{-s} \tag{1}$$

Причем в разобранных примерах s - и есть то, что мы интуитивно считаем размерностью объекта, причем она совпадает с размерностью самого пространства.

Выделим s из формулы (1): $s = \frac{\ln N}{-(\ln c + \ln \delta)}$.

Теперь заметим, что формула (1) в двух разобранных примерах выполняется точно лишь при удачном выборе числа δ. Вообще говоря, не при каждом числе δ эта формула будет выполняться. Но этого и не надо. Главное, чтобы она была верна при $\delta \to 0$, т.е. когда мы будем покрывать очень малыми объектами. Поэтому введем такое определение размерности:

- Размерностью множества Q назовем величину s, вычисляющуюся по формуле $s = \lim_{\delta \to 0} \frac{\ln N}{-(\ln \delta)}$, где $N = N(\delta)$ - минимальное количество квадратов с длиной стороны не больше δ, которыми можно покрыть множество Q.

Вообще говоря, покрывать можно не обязательно квадратами. Можно покрывать, например, кругами, размерность от этого не изменится, хотя коэффициент c в формуле (1) будет иной.

Чтобы доказать это достаточно заметить, что в круг любого радиуса можно вписать квадрат и около этого круга можно описать квадрат:

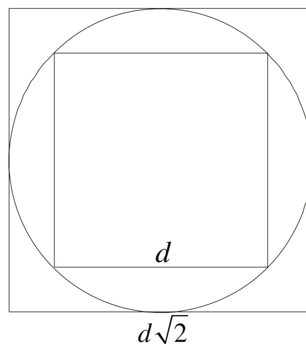


Рис 18.4.

Введем обозначения:

$N_Q(x)$ - минимальное количество квадратов с длиной стороны не больше x, которыми надо покрыть заданное множество.

$N_K(x)$ - минимальное количество кругов диаметром не больше x, которыми надо покрыть заданное множество.

Очевидно, что:

$$N_Q(\delta) \le N_K(\delta\sqrt{2}) \le N_Q(\delta\sqrt{2})$$

При $0<\delta<1$ получим: $\dfrac{\ln\left(N_Q(\delta)\right)}{-\ln\delta} \le \dfrac{\ln\left(N_K(\delta\sqrt{2})\right)}{-\ln\delta} \le \dfrac{\ln\left(N_Q(\delta\sqrt{2})\right)}{-\ln\delta}$.

Заметим, что $\lim\limits_{\delta\to 0}\dfrac{\ln\left(N_Q(\delta\sqrt{2})\right)}{-\ln\delta} = \lim\limits_{\delta\to 0}\dfrac{\ln\left(N_Q(\delta\sqrt{2})\right)}{-\ln(\delta\sqrt{2})+\ln\sqrt{2}} = \lim\limits_{\delta\to 0}\dfrac{\ln\left(N_Q(\delta\sqrt{2})\right)}{-\ln(\delta\sqrt{2})} = \lim\limits_{\delta\to 0}\dfrac{\ln\left(N_Q(\delta)\right)}{-\ln(\delta)}$.

Точно так же получим следующее соотношение: $\lim\limits_{\delta\to 0}\dfrac{\ln\left(N_K(\delta\sqrt{2})\right)}{-\ln\delta} = \lim\limits_{\delta\to 0}\dfrac{\ln\left(N_K(\delta)\right)}{-\ln(\delta)}$.

Значит $\lim\limits_{\delta\to 0}\dfrac{\ln\left(N_Q(\delta)\right)}{-\ln(\delta)} = \lim\limits_{\delta\to 0}\dfrac{\ln\left(N_K(\delta)\right)}{-\ln(\delta)}$, т.е. размерность не зависит от того, считаем ли мы кругами диаметра δ или квадратами со стороной δ.

Давайте теперь воспользуемся таким определением для того, чтобы подсчитать размерность снежинки Коха (см. главу «Рекурсия»). Покрывать снежинку для удобства будем кругами диаметра δ. Пусть сторона исходного треугольника будет равна 1 ед.

Если $\delta=\dfrac{1}{3}$, то количество кругов равно $3\cdot 4$ (на рис. 18.5 показано, как покрывается одна сторона снежинки). Ясно, что если $\delta=\dfrac{1}{3^k}$, то количество кругов равно $3\cdot 4^k$.

Теперь давайте считать размерность:

$$s=\lim_{\delta\to 0}\frac{\ln N}{-(\ln\delta)} = \lim_{k\to\infty}\frac{\ln(3\cdot 4^k)}{-(\ln 3^{-k})} = \lim_{k\to\infty}\frac{\ln(3)+k\ln 4}{k\ln 3} = \frac{\ln 4}{\ln 3} = \log_3 4\,.$$

Выходит, что если ввести размерность по формуле (2), то действительно существуют множества дробных размерностей.

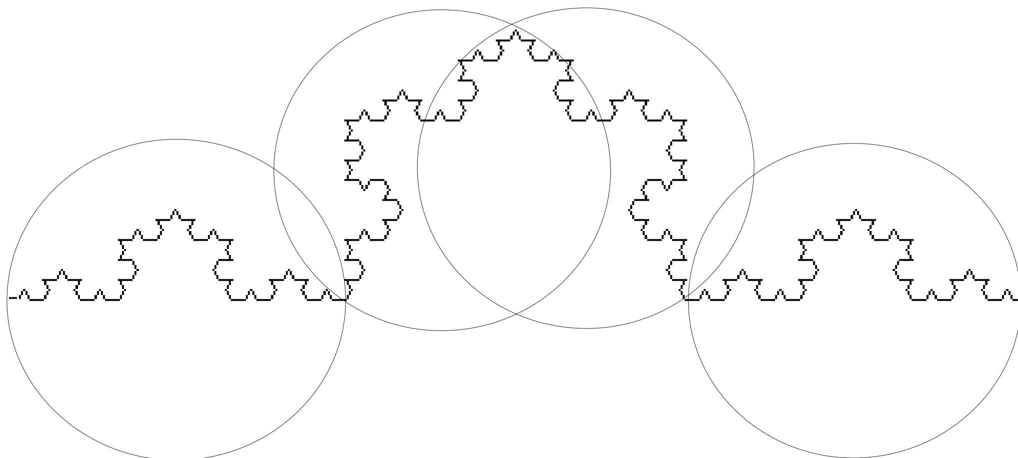


Рис. 18.5. Покрытие снежинки Коха кругами

Вообще говоря, существуют различные определения размерности, причем то, которое мы дали (размерность Минковского) – далеко не самое общее, причем оно обладает рядом недостатков. Например, некоторые счетные[18] множества могут иметь ненулевую размерность. Существует определение размерности, которое дал Феликс Хаусдорф, которое не содержит подобных недостатков, однако размерность Хаусдорфа вычислять значительно труднее. Однако во многих случаях, в частности, для самоподобных фигур, как снежинка Коха, размерности Хаусдорфа и Минковского совпадают.

Фрактальной структурой обладают, например, корни растений: суммарная длина корней одной особи ржи = 622,8 км, а площадь поверхности корней = 640 $м^2$.

Теперь мы можем приступить к написанию программы, которая могла бы строить фракталы (приближенно, разумеется). Для этого рассмотрим понятие L-систем.

18.6. Построение самоподобных фракталов

Идея построения фракталов очень проста. Для того, чтобы рисовать изображения можно использовать так называемую черепашью графику. Черепаха характеризуется положением на экране (x, y) и углом поворота относительно оси ox - θ. Черепаха может двигаться вперед на некоторое расстояние r или поворачиваться на месте вправо или влево на угол α.

Для того чтобы черепаха могла нарисовать что-то на экране, надо написать программу на черепашьем языке (ЧЯ). Я буду придерживаться обозначений, принятых в книге Ричарда М. Кроновера «Фракталы и хаос в динамических системах. Основы теории»:

F	продвинуться вперед на расстояние r, прорисовывая след
b	продвинуться вперед на расстояние r, не прорисовывая след
[	открыть ветвь
]	закрыть ветвь
+	повернуться вправо на угол α
-	повернуться влево на угол α

Открыть ветвь означает сохранить положение черепахи (т.е. координаты и направление движения). Закрыть ветвь – это означает переместиться в точку, где находилась черепаха в момент, когда ветвь была открыта (т.е. вернуться к координатам, которые были сохранены в момент открытия ветви).

Итак: черепаха получает слово, состоящее из вышеперечисленных символов (программу на черепашьем языке), и выполняет содержащиеся в этом слове действия.

Теперь дело за малым – как построить слово, которое бы представляло собой пример фрактального изображения (т.е. некоторого приближения фрактала, т.к. бесконечное количество итераций отобразить нельзя).

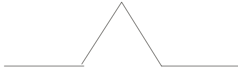
Мы будем строить фракталы, которые начинаются из некоторой базовой фигуры, а потом на каждой итерации изменяются по определенным правилам, которые нам известны. Одним из таких фракталов является снежинка Коха.

---

[18] Множество называется счетным, если его можно биективно отобразить на множество натуральных чисел.

Т.к. есть базовая фигура, значит, можно ее записать на черепашьем языке. Назовем эту строку аксиомой. Для острова Коха аксиома – равносторонний прямоугольник, который на ЧЯ записывается так: F++F++F (величина угла равна $\frac{\pi}{3}$).

На каждой итерации прямая заменяется следующей фигурой:



Это означает, что вместо буквы F надо подставить строку, которая задает эту фигуру: F-F++F-F.
Эту строку будем называть подстановкой и обозначаться она будет в алгоритме neuf (neu - новый). Аналогично вместо b будет подставляться neub.
Например, остров Коха после первой итерации будет задаваться такой строкой:
F-F++F-F++F-F++F-F++F-F++F-F

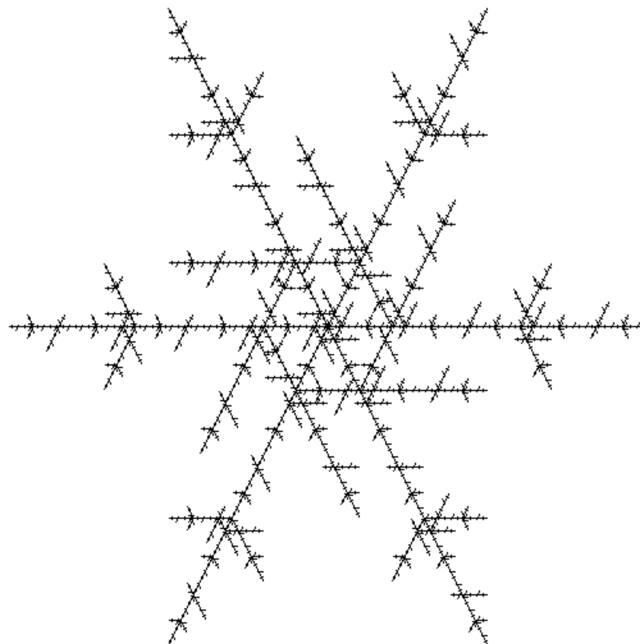


Рис 18.6. Снежинка (axiom = [F]+[F]+[F]+[F]+[F]+[F], neuf = F[++F][-FF]FF[+F][-F]FF)

Очевидно, что множество Кантора задается следующими параметрами:
ax = F
neuf = FbF
neub=bbb

Т.е. алгоритм построения фракталов таков: берем аксиому, затем вместо F подставляем в нее neuf, а вместо b подставляем neub.

18.7. Реализация черепашьей графики

Теперь мы напишем программу, реализующую предложенный алгоритм.

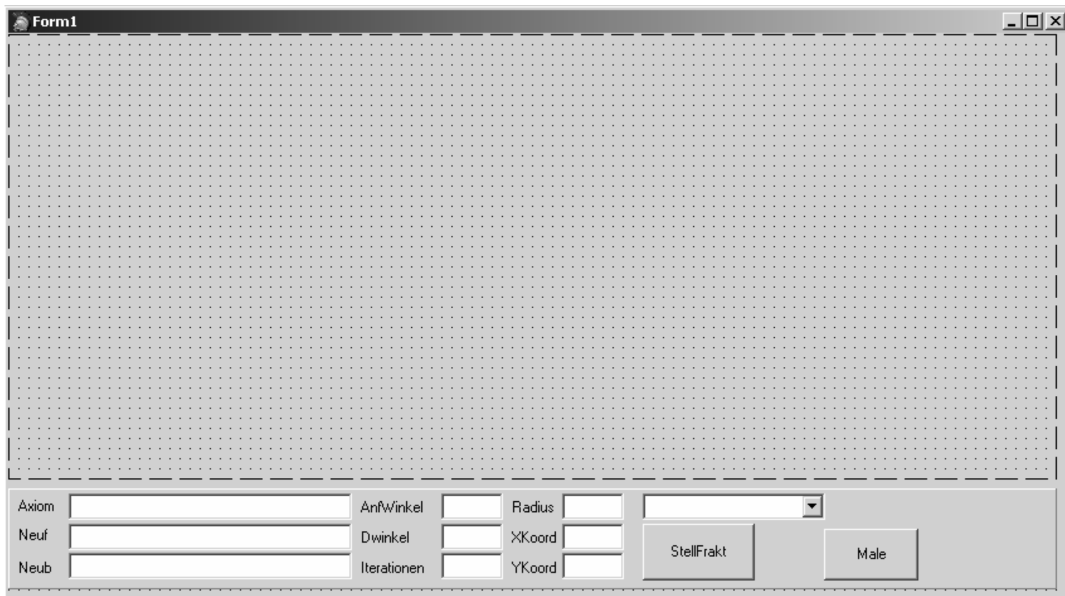


Рис. 18.7. Форма для примера по рисованию фракталов

Форма разделена на 2 части: место для рисования (компонент PaintBox типа TPaintBox во вкладке System) и панель (компонент Panel типа TPanel во вкладке Standard). На панель можно ставить компоненты также, как и на форму, причем их координаты устанавливаются относительно левого верхнего угла панели. У класса TPaintBox есть канва, на которой мы и будем рисовать. Можно было бы рисовать и на канве формы, но это не так удобно.

9 текстовых полей – это параметры, которые надо задать, чтобы получить фрактал. В Комбобоксе можно будет выбирать готовые фракталы (при этом будут заполняться лишь поля ввода, в которых должны находиться следующие параметры: axiom, neuf, neub, AnfWinkel, Dwinkel). Остальные параметры надо будет вводить самостоятельно.

Основных классов 3: SchildKröte (черепаха), FraktMacher (создатель фракталов), FraktMaler («рисовальщик фракталов»).

Fraktmaler – это класс, который в качестве своих компонентов содержит Schildkröte и FraktMacher. Он создан просто для удобства и лучшей структурированности.

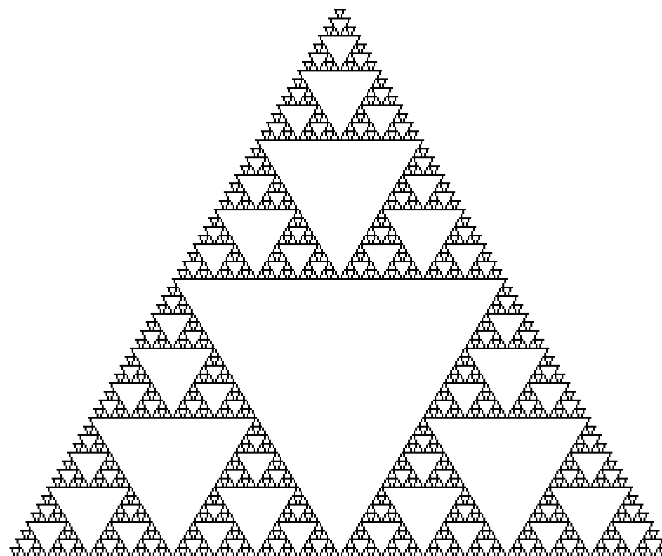


Рис 18.8. Ковер Серпинского (axiom = F++F++F, Neuf = F--F++F++F--F)

Пример 3: Черепашья графика.

Я приведу в книге лишь 2 основные процедуры из этой программы:

1. Генератор фракталов GibFrakt (класса FraktMacher)

```
function FraktMacher.GibFrakt:string;
var
  z:string;
  i,j:integer;
begin
  if n=0 then
    begin
    result:=ax;
    exit;
    end;
  z:=ax;
  for i:=1 to n do
    begin
    result:='';
    for j:=1 to length(z) do
      case z[j] of
      'F': result:=result+nf;
      'b': result:=result+nb;
      else
        result:=result+z[j];
      end;
    z:=result;
    end;
end;
```

1. Выполнение программы на черепашьем языке LeisteProgr (класса SchildKröte)

```
procedure SchildKrote.LeisteProgr(s:string;C:TCanvas);
var
  i,k:integer;
  Arr:array of Lage;
  AnfLage:Lage;
  rwink:real;
begin
  AnfLage:=self.SKLage;    //сохраняем текущее положение черепахи

C.MoveTo(L.x,L.y);
SetLength(Arr,100);
k:=0;

for i:=1 to length(s) do
  case s[i] of
  'F': begin
       with L do
         begin
         rwink:=ZuRadian(L.wink);
```

```
          x:=round(x+r*cos(rwink));
          y:=round(y+r*sin(rwink));
          C.LineTo(x,y);
          end;
        end;
   '+': L.wink:=L.wink+dteta;
   '-': L.wink:=L.wink-dteta;
   'b': begin
        with L do
        begin
        rwink:=ZuRadian(L.wink);
        x:=round(x+r*cos(rwink));
        y:=round(y+r*sin(rwink));
        C.MoveTo(x,y);
        end;
        end;
   '[': begin
        Arr[k]:=L;//загоняем в стек позицию черепахи
        k:=k+1;
        end;
   ']': begin
        L:=Arr[k-1]; //вытаскиваем их стека позицию черепахи
        C.MoveTo(L.x,L.y);
        k:=k-1;
        end;
  end;
  self.SKLage:=AnfLage; //устанавливаем исходное положение черепахи
end;
```

Остальные процедуры, которые находятся вне основного модуля, имеют чисто вспомогательное значение. Что касается скорости работы этих двух алгоритмов, то скажу, что алгоритм построения черепашьего слова, который приведен выше, очень неэффективен. Можно легко придумать гораздо более быстрый алгоритм, что и предлагается вам сделать в упражнении 11.

Если посмотреть на алгоритм обработки черепашкой программы на ее языке, то видно, что единственной сложной математической операцией является вычисление косинуса и синуса. Поэтому для фракталов, у которых угол, на который изменяется движение черепахи, можно представить в виде $\frac{\pi}{n}$, где n - целое число, отличное от нуля, можно ускорить алгоритм отрисовки фрактала, просто заранее вычислив все возможные значения cos(rwink), sin(rwink). Тогда объем вычислений сократится во много раз. Можете воспользоваться этим, чтобы ускорить алгоритм отрисовки черепахой фрактала.

Готовые фракталы, которые можно выбирать в комбобоксе, хранятся в файле 'FertigenFraktalen.txt' (т.е. готовые фракталы). Они хранятся в следующем формате:

Название фрактала
Axiom
Neuf
Neub

AnfWinkel
DWinkel

Вы можете, естественно, дописывать в файл собственные фракталы, и они будут в начале работы программы (см. обработчик FormCreate события формы OnCreate) загружаться из этого файла в комбобокс.

Фактически в основном модуле ничего особенного кроме подпрограммы, которая считывает фракталы из файла и обработчиков разных событий, нет.

Задачи

1. Создайте класс Damestein (шашка), напишите основные методы.
2. Создайте наследника класса Brett (SpielBrett – «доска для игры»), в котором изменится процедура рисования: кроме самой доски должна выводиться аннотация (т.е. координатная система). Систему координат возьмите из русских или стоклеточных шашек.
3. Напишите класс DameSpielBrett, наследника SpielBrett, в котором кроме информации о доске находились бы массивы шашек. У класса DameSpielBrett должен быть метод, позволяющий расставлять эти массивы шашек на доску.
4. Напишите класс KoordSys, в котором бы находилась вся необходимая информация для построения системы координат на плоскости. Нарисуйте на этой системе координат график функции $y = \sin x$ и усеченные суммы ряда $y(x) = \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}$, т.е. $y_1 = x$, $y_2 = x - \frac{x^3}{3!}$, $y_3 = x - \frac{x^3}{3!} + \frac{x^5}{5!}$ и т.д., и посмотрите, насколько точно ряд соответствует настоящим значениям синуса.
5. В главе рекурсия мы рассматривали задачу: как расставить n ферзей на доске размером $n \times n$ так, чтобы они не били друг друга. Напишите программу, которая бы не только находила подходящие комбинации, но и некоторые из них (по желанию пользователя) рисовала на доске SpielBrett (см. задачу №2).
6. Тело веса P, брошенное с начальной скоростью v_0 под углом α к горизонту, движется под влиянием силы тяжести. Найти уравнения движения тела (считая, что тело начинает движение из начала координат), и начертить график, используя класс KoordSys. Можете при желании расширить класс KoordSys, пополнив его методами вычерчивания графиков.
7. **(!)** Решить предыдущую задачу, если на тело действует кроме силы тяжести еще и сопротивление воздуха $R = kPv$, k - коэффициент пропорциональности тела. Нарисуйте график движения тела.
8. **(!)** Тело начальной массы m_0 начинает движение из начала координат, и движется под действием силы тяжести и реактивной силы. Пусть масса топлива составляет $m_0 / 2$, закон изменения массы ракеты задается выражением: $m(t) = m_0 e^{-kt}$, где k - некоторый коэффициент (например, 0.005), скорость истечения газов постоянна, направлена под углом α к горизонту и равна $u = 2000 \frac{м}{с}$. Учтите, что когда топливо заканчивается, тело продолжает двигаться

под действием лишь силы тяжести. Найдите закон, по которому движется тело, и время движения тела до падения на землю. Начертите график, и сравните его с графиками, которые получились в предыдущих двух задачах.

9. Вычислите размерность множества Кантора.
10. Вычислите размерность пыли Кантора.
11. Алгоритм построения фракталов, использованный для построения слова для черепахи, очень медленный. Придумайте как можно более быстрый алгоритм генерации черепашьего слова.
12. Постройте аналог черепашьей графики для того, чтобы можно было бы рисовать сплошные фрактальные области.

Проект 6: Беовульф и Нибелунги

От криптографии перейдем к лингвистике. Одной из распространенных задач лингвистов и историков является расшифровка древних языков. Криптографы изначально предполагают, что противнику, который пытается расшифровать текст, может быть известен и метод кодировки, и тема сообщения, и язык, на которым оно написано, а не известен лишь ключ – в случае подстановочного шифра – это конкретная подстановка, с помощью которой был закодирован текст. У специалистов по древним языкам ситуация иная: алгоритм шифрования (т.е. язык) неизвестен. Поэтому надо использовать все данные о самом народе, чей язык изучается и сведения о родственных языках (если такие существуют).

Расшифровка значительно упрощается, если к текстам, написанным на неизвестном языке, есть переводы на другие языки (хотя бы 1). Разумеется, наличие таких текстов тоже совсем не гарантирует успеха.

Вам на рассмотрение я предлагаю следующую задачу:

У вас есть текст древнеанглийской эпической поэмы «Беовульф» и переводы его на 3 языка: английский, немецкий и русский. Я предполагаю, что древнеанглийским вы не владеете, но при этом кроме русского знаете либо немецкий, либо английский.

Древнейшая из найденных рукописей «Беовульфа» датируется X веком н.э., однако считается, что эта поэма была написана в 8 веке н.э. В поэме повествуется о подвигах Беовульфа, конунга гаутов, – как он боролся с зломерзкими тварями на земле и на море, правил землями и защищал свой народ от нападений соседей.

Ниже вы можете прочитать первые 25 строк «Беовульфа» на староангл., англ., нем., русск. языках. Полный текст (английский и, в особенности, немецкий текст содержат комментарии к тексту).

Беовульф

Староанглийский

Hwæt! Wé Gárdena in géardagum
þéodcyninga þrym gefrúnon·
hú ðá æþelingas ellen fremedon.
Oft Scyld Scéfing sceaþena þréatum
monegum maégþum meodosetla ofteáh·
egsode eorle syððan aérest wearð
féasceaft funden hé þæs frófre gebád·
wéox under wolcnum· weorðmyndum þáh
oð þæt him aéghwylc þára ymbsittendra
ofer hronráde hýran scolde,
gomban gyldan· þæt wæs gód cyning.
Ðaém eafera wæs æfter cenned
geong in geardum þone god sende
folce tó frófre· fyrenðearfe ongeat·
þæt híe aér drugon aldorléase
lange hwíle· him þæs líffreä
wuldres wealdend woroldáre forgeaf:
Béowulf wæs bréme --blaéd wíde sprang--

Современный английский

LO, praise of the prowess of people-kings
of spear-armed Danes, in days long sped,
we have heard, and what honor the athelings won!
Oft Scyld the Scefing from squadroned foes,
5 from many a tribe, the mead-bench tore,
awing the earls. Since erst he lay
friendless, a foundling, fate repaid him:
for he waxed under welkin, in wealth he throve,
till before him the folk, both far and near,
10 who house by the whale-path, heard his mandate,
gave him gifts: a good king he!
To him an heir was afterward born,
a son in his halls, whom heaven sent
to favor the folk, feeling their woe
15 that erst they had lacked an earl for leader
so long a while; the Lord endowed him,
the Wielder of Wonder, with world's renown.
Famed was this Beowulf: far flew the boast of him,

Scyldes eafera Scedelandum in.
Swá sceal geong guma góde gewyrcean
fromum feohgiftum on fæder bearme
þæt hine on ylde eft gewunigen
wilgesíþas þonne wíg cume·
léode gelaésten: lofdaédum sceal
in maégþa gehwaére man geþéön.

son of Scyld, in the Scandian lands.
20 So becomes it a youth to quit him well
with his father's friends, by fee and gift,
that to aid him, aged, in after days,
come warriors willing, should war draw nigh,
liegemen loyal: by lauded deeds
25 shall an earl have honor in every clan.

Немецкий

Hört! Denkwürd'ger Taten von Dänenhelden
Ward uns viel fürwahr aus der Vorzeit berichtet,
Wie Könige kühn ihre Kraft erprobten.
Der Garbensohn Scyld hat oft grimme Feinde,
Viel mutige Krieger vom Metsitz verjagt 5
Und Furcht verbreitet. In früher Jugend
Fand man hilflos ihn auf, doch Heil ersproß ihm:
Unterm Wolkendach wuchs er, an Würden reich,
Bis alle endlich ihm untertan wurden,
Die am Wege des Wals ihren Wohnsitz hatten, 10
Und Zins dem Herrlichen zollen mußten.
Ein Sprößling ward ihm später geboren,
Ein holder Knabe, vom Herrgott gesendet
Dem Lande zum Trost: das Leid erbarmt' ihn,
Das die Dänen lange erduldet hatten, 15
Eines Oberhaupts ledig. Dem Erben Scylds
Verlieh der leuchtende Lebensspender
Blühende Ehren und Beowulfs Ruhm
Erscholl weithin in Schonens Gauen.
So schenke in jungen Jahren der Mann 20
Vom Hort freigebig im Hause des Vaters,
Daß willig im Alter ihn wiederum stützen
Die kühnen Kämpen, wenn Krieg entbrennt,
Und mutig ihm folgen: die milde Hand
Wird überall dem Edling frommen. 25

Русский

Истинно! исстари слово мы слышим
о доблести данов, о конунгах датских
чья слава в битвах мечами добыта!
Первый – Скильд Скевинг, войсководитель,
не раз отрывавший вражьи дружины
от скамей бражных. За все, что он выстрадал
в детстве, найденыш, ему воздалось:
стал разрастаться властный под небом
и, возвеличенный, силой принудил
народы заморья дорогой китов
дань доставить достойному власти
Добрый был конунг! В недолгом времени
сын престола, наследник родился,
посланный Богом людям на радость
и в утешение, ибо Он видел
их гибель и скорби в век безначалия, -
от Вседержителя вознаграждение,
 от жизнеподателя благонаследие;
знатен был Беовульф, Скильдово семя,
в датских владениях. С детства наследник
добром и дарами дружбу дружины
должен стяжать, дабы, когда возмужает,
соратники стали с ним о бок,
верные долгу, если случится война, -
ибо мужу должно достойным
делом в народе славу снискать!

Ваша цель – как можно дальше продвинуться в расшифровке древнеанглийского текста.

Написать программу, которая смогла бы сама установить значение всех или большинства слов неизвестного языка – очень сложно. Но это и не требуется. Часто случается, что решение некоторой задачи приводит к появлению огромного числа идей, которые сами по себе очень важны. Расшифровка неизвестных языков – одна из таких задач.

С чего начинать писать программу?

1. ПК может быстро составить список слов для всех 4-х текстов. Разумеется, перевод далеко не дословный, однако частотность появления многих слов будет примерно одинакова.
2. Можно использовать «Поиск сходных слов», причем двумя способами:

- Искать слова, которые сходны в текстах на разных языках (можно учитывать при этом и их положение в тексте)
- Искать словоформы одного и того же слова в неизвестном языке. Если текст достаточно большой (в «Беовульфе» свыше 3000 строк) то, быть может, удастся установить некоторые правила грамматики.

Примеры схожести слов из разных языков:

æþelingas - athelings – Edlingen
wolcnum – welkin – Wolken
sende – sent – gesendet
fæder – father - Vater

В одном языке:

geardagum - geardum

Даже сама частота встречаемости слов может уже сказать о многом. Например, артикли, которые есть в немецком и английском языках, отсутствуют в русском (сразу напрашивается вопрос, есть ли они в древнеанглийском).

Фактически, от вас как от программиста требуется создать как можно лучшего помощника лингвиста. Затем, я думаю, вам должно быть интересно самому выступить в роли лингвиста и попытаться разобраться в тексте. Проект не должен быть привязан конкретно к староанглийскому языку: желательно, чтобы вы могли бы без переписывания большей части программного кода применить программу для расшифровки текстов на других языках.

Можете, если хотите, вместо Беовульфа взять другой эпос, например «Песнь о Нибелунгах»:

Песнь о Нибелунгах, современный Hochdeutsch

Viel Wunderdinge melden | die Mären alter Zeit
Von preiswerthen Helden, | von großer Kühnheit,
Von Freud und Festlichkeiten, | von Weinen und von Klagen,
Von kühner Recken Streiten | mögt ihr nun Wunder hören sagen.

Es wuchs in Burgunden | solch edel Mägdelein,
Daß in allen Landen | nichts Schönres mochte sein.
Kriemhild war sie geheißen, | und ward ein schönes Weib,
Um die viel Degen musten | verlieren Leben und Leib.

Die Minnigliche lieben | brachte Keinem Scham;
Um die viel Recken warben, | Niemand war ihr gram.
Schön war ohne Maßen | die edle Maid zu schaun;
Der Jungfrau höfsche Sitte | wär eine Zier allen Fraun.

Es pflegten sie drei Könige | edel und reich,
Gunther und Gernot, | die Recken ohne Gleich,
Und Geiselher der junge, | ein auserwählter Degen;
Sie war ihre Schwester, | die Fürsten hatten sie zu pflegen.

Die Herren waren milde, | dazu von hohem Stamm,
Unmaßen kühn nach Kräften, | die Recken lobesam.
Nach den Burgunden | war ihr Land genannt;

Sie schufen starke Wunder | noch seitdem in Etzels Land.

Песнь о Нибелунгах, средневерхненемецкий

In alten mæren wnd*er*s vil geseit
von heleden lobebæren vo*n* grozer arebeit
von frevde vn– hochgeciten von weinen vn– klagen
von kvner recken striten mvget ir nv wnd*er* horen sagen

Ez whs <inBvregonden> ein vil edel magedin
daz in allen landen niht schon*er*s mohte sin
Chriemhilt geheizen div wart ein schone wip
dar vmbe mvsin degene vil v*er*liesen den lip

Ir pflagen dri kunige edel un– rich
Gunther un– Gernot die rechen lobelich
vn– Giselher d*er* iunge ein wetlich*er* degen
div frowe was ir swester die helde hetens inir pflegen

Ein richiv chuniginne frov Vte ir mvt*er* hiez
ir vat*er* d*er* hiez Dancrât d*er* in div erbe liez
sit nach sime lebene ein ellens rich*er* man
d*er* ovch insiner iugende grozer eren vil gewan

Die h*er*ren waren milte von arde hoh erborn
mit kraft vn– mazen chvne die rechen vz erchorn
da zen Bvrgonden so was ir lant genant
si frvmten starchiv wnder sit in Etzelen lant

Работа с текстовым файлом

Тексты поэмы приводятся в html-формате, поэтому если вы его не знаете, то вам придется разобраться с тем, как писать html-файлы. Не пугайтесь, ничего сложного в этом нет.

Сначала вам придется преобразовать текст из html-формата в какой-то более подходящий. Но учтите, что в древнеанглийском тексте есть буквы, которые не входят во множество символов ASCII, поэтому либо вы должны каждой букве древнеанглийского текста поставить в соответствие некоторый не встречающийся в тексте символ, либо попытаться перевести html-файл в текстовый файл на основе кодировки Unicode (двухбайтовые целые).

Думаю, что эти технические сложности вас не сильно затруднят, т.к. решить эти проблемы значительно проще, чем заниматься расшифровкой «Беовульфа» или «Нибелунгов».

Глава 19: Проблемы ООП. Везенспрограммирование

19.1. История языков программирования

На протяжении всей книги мы изучали императивные языки – как процедурные, так и объектно-ориентированные. Теперь пришла пора оценить положение, в котором сейчас находится теория языков программирования. В следующей таблице приведены основные события в истории языков программирования.

Год	Событие	Характеристика языка
1941	Конрад Цузе в Германии построил машину Z3 на основе электромеханических реле	
1943	В Англии построена первая электронная ЭВМ – Colossus.	
1944	Джон Эккерт выдвинул концепцию программы, хранимой в памяти	
1945	В Америке построена ЭВМ ENIAC	
1949	**Первый язык ассемблера**	**Ассемблер**
1951	**Грейс Хоппер разработала первый транслятор для программы, записанной в удобной алгебраической форме**	
1955	**Fortran**	**Процедурный**
1958	ALGOL 58	Процедурный
1959	**LISP**	**Функциональный**
1961	**GPSS**	**Логический (декларативный)**
1967	**Simula 67**	**ОО-расширение императивных языков**
1970	Prolog	Логический (декларативный)
1971	Pascal	Процедурный
1972	C	Процедурный
1972	Smalltalk 72	ОО-расширение императивных языков
1981	1-й персональный компьютер (IBM PC)	
1986	C++	ОО-расширение императивных языков
1986	Object Pascal	ОО-расширение императивных языков
1988	CLOS	ОО-расширение LISP
1996	Java	ОО-расширение императивных языков
2002	C#	ОО-расширение императивных языков

Как видите, все концепции программирования, которые используются на сегодняшний день, появились до 1967 года. Через 22 года после появления машины ENIAC, которая работала в миллионы раз медленнее, чем современные ПК, которая постоянно ломалась и занимала целые комнаты, уже были сформулированы все концепции программирования, используемые на сегодняшний день, и написаны

компиляторы для языков, реализующих эти идеи. В следующие 40 лет радикально новых концепций выдвинуто не было (имеются в виду именно языки программирования, - сети и аппаратное обеспечение не в счет) – лишь развивались уже существующие, несмотря на то, что количество программистов и производительность компьютеров увеличились на порядки.

19.2. Обзор некоторых объектно-ориентированных языков

Чтобы оценить современное состояние ООП, мы рассмотрим вкратце основные ОО-языки.

Simula

Первый объектно-ориентированный язык – Simula 67 - появился еще в 1967 году. Он был создан Кристеном Нигардом (Kristen Nygaard) и Оле-Йоханом Далом (Ole-Johan Dahl) из Норвежского Вычислительного Центра (Norsk Regnesentral) и Университета Осло. В 1986 году Simula 67 был переименован в Simula.

Simula 67 – наследник языка Simula 1, созданного в начале 60-х годов для моделирования дискретных событий. В отличие от своего предшественника, Simula 67 – это многоцелевой ОО-язык. Оба языка являются расширениями языка ALGOL 60 – одного из самых популярных языков в то время.

Класс в Simula состоит из полей, методов и тела класса.

Тело класса в чем-то заменяет конструктор класса. При создании объекта выполняются операторы тела класса.

В Simula реализованы все основные понятия ООП – наследование, полиморфизм, абстрактные классы.

Кроме того, в Simula есть понятие сопрограмм (мы его рассматривали в главе рекурсия). Сопрограммы реализованы на основе классов. С их помощью разработчики Simula добились аналога параллельности работы (хотя сопрограммы – это не параллелизм, т.к. они предусматривают жесткую зависимость сопрограмм друг от друга). Позднее, когда появились мультипотоковые ОС, поддержку параллельности перенесли в них.

Все ОО-языки – потомки Simula 67, поэтому значение языка поистине огромно.

Simula – не мертвый язык – он по сей день используется для разработки ПО (в основном в Скандинавии).

Smalltalk

Вторым ОО-языком стал Smalltalk. Базовые концепции ООП Smalltalk позаимствовал из Simula, но в то же время в этом языке вводится много новых понятий:

- В Smalltalk все переменные – объекты (даже числа).
- Есть базовый класс - object.
- Сами классы – объекты метаклассов.
- У классов есть собственные методы.
- Поля классов могут быть только закрытыми.

В Smalltalk нет статической проверки типов, поэтому все необходимые проверки применимости операций проводятся во время работы программы. Именно это

обстоятельство привело к тому, что производительность программ, написанных на Smalltalk, относительно низка. Но это не умаляет значения Smalltalk для ООП: метаклассы и единая иерархия классов – важнейшие нововведения. С помощью этих понятий можно легко добиваться общности программ.

Delphi (Object Pascal)

Известно много различных объектно-ориентированных расширений для языка Pascal. Мы дадим оценку лишь для Delphi, который мы рассмотрели достаточно подробно.

Delphi строился как расширение языка Turbo Pascal и первоначально назывался Object Pascal. Поэтому Delphi позволяет писать программный код как в процедурном, так и в ОО-стиле.

Delphi многое перенял от Smalltalk. То, что все классы происходят от TObject разумно и в практическом смысле, т.к. позволяет писать методы, которые будут общими для объектов любых классов и в концептуальном, т.к. в любом случае каждый объект содержит по крайней мере ссылку на себя и ссылку на дескриптор класса, поэтому сделать класс, объекты которого содержат эти поля и еще несколько базовых методов, представляется довольно разумным.

Использование ссылок позволяет обойтись без указателей, которые используются в процедурном программировании.

В Delphi есть все базовые возможности ООП:

- инкапсуляция с делением полей на private, public, protected.
- Одиночное наследование
- Полиморфизм, реализованный с помощью динамического связывания

В Delphi очень естественно реализована возможность получения информации о типе объекта во время выполнения (RTTI – Run Time Type Information), - в базовом классе TObject можно получать ссылки на дескриптор класса.

Метаклассы и операторы is, as позволяют с легкостью орудовать на уровне классов, а не объектов, поэтому легко достигается общность программ.

Интерфейсы позволяют сделать более структурированной иерархию классов без использования множественного наследования (см. раздел о С++).

Свойства реализованы довольно громоздко, но в принципе идея неплохая.

Недостатки:

1. Возможность доступа к private-членам класса в других классах, реализованных в том же модуле. Это – пережиток модульного прошлого Delphi. Чтобы избавиться от этого недостатка, в Delphi 2005 ввели определители видимости элементов strict private, strict protected.
2. Деструктор ничем особым не выделяется от других методов, поэтому можно было бы обойтись без дополнительного ключевого слова.
3. Большой набор базовых функций в классе TObject: многие из них являются подметодами для других методов, и сами по себе нужны крайне редко.
4. Думаю, что разработчикам Delphi не стоило стремиться к возможно более полной совместимости с ТР. Из-за этого Delphi перенял оператор goto и тип object, который устарел т.к. были введены классы. Можно вспомнить также такое понятие, как «типизированная константа», которая на самом деле – переменная, хотя и

объявляется в разделе const. Зачем было называть переменную, инициализированную значением типизированной константой, непонятно. Все эти недостатки справедливо вызывают нарекания у многих программистов.
5. Нет переменных класса

Сложнее вопрос, нужны ли указатели и встроенный ассемблер (мы его не рассматривали). С одной стороны, т.к. в Delphi иерархия классов идет от TObject и работа с объектами ведется только при помощи ссылок, то указатели, по большому счету, для написания ОО-приложений не нужны. С другой стороны, указатели и встроенный ассемблер могут очень помочь, если требуется достичь большей эффективности приложений - например, для написания компиляторов.

На мой взгляд, указатели и встроенный ассемблер убирать не надо, т.к. процедурный и ОО-подходы в Delphi не мешают друг другу: не нужны указатели, так и не используйте их. Вас никто не заставляет это делать.

Эти слова относились к Delphi 7. В Delphi 2005, стремясь подстроиться под платформу .NET, был введен целый ворох средств, которые не приносят ничего принципиально нового, но при этом загромождают язык. Тем не менее, как ни старались разработчики Delphi, перещеголять С++ по количеству ненужных нововведений им не удалось.

С++

С++ - один из наиболее популярных ОО-языков, созданный в 1986 году Бьярном Страуструпом. Он представляет собой расширение языка С, разработанного Деннисом Ритчи для разработки ОС Unix в 1972 году.

Язык С создавался для системного программирования – он позволил программистам меньше кода писать на ассемблере, что их несомненно радовало. В С есть механизмы, которые вообще не являются процедурным программированием – например, макросы. В TP их нет, поэтому давайте рассмотрим это понятие более подробно.

Макросы

- Макросы - параметризованные символические константы.

Перед началом компиляции компилятор заменяет вхождения макроса на строку, вставляя в нее параметры.

Макросы можно создавать с помощью директивы `#define`. В примере вводится 3 макроса `aPb1, aPb2, aPb3`.

Пример 1: Макросы в С++.

```
#include <iostream.h>;  //подключение библиотеки iostream.h

#define aPb1(a,b) (a*b)
#define aPb2(a,b) a*b
#define aPb3(a,b) ((a)*(b))

void main()  //основная подпрограмма
{
  double c = (double)10/aPb1(1+2,4);   //c=10/(1+2*4)
```

```
  double d = (double)10/aPb2(1+2,4);  //d=10/1+2*4
  double e = (double)10/aPb3(1+2,4);  //e=10/((1+2)*4)
  cout<<"c ="<<c<<" d ="<<d<<" e "<<e;
}
```

Например, разберем определение макроса `aPb1`:

`#define aPb1(a,b) (a*b)`

`aPb1(a,b)` – заголовок макроса

`(a*b)` – значение, на которое макрос будет заменен.

В частности, aPb1(1+2,3+4) преобразуется в (1+2*3+4).

В теле программы демонстрируется различие между тремя макросами, которые описаны в программе. Из трех макросов лишь `aPb3` правильно реализует умножение чисел.

(double) – это операция приведения типов. Дело в том, что отношение двух целых чисел в С – целое число, поэтому без приведения типов не обойтись.

Пример, который я привел – далеко не самый хитрый. Например, в каком порядке будет компилятор выполнять замены в следующем выражении: aPb3(aPb3(1,2),3)?

Ответ: вначале будет подставлена строка вместо вложенного макроса, а потом будет заменен внешний макрос, т.е. результат будет такой:

aPb3(aPb3(1,2),3) -> aPb3(((1)*(2)),3) -> ((((1)*(2)))*(3))

Поэтому для работы с макросами мало знать, что это такое, - надо знать еще дополнительные правила, одно из которых - очередность подстановки.

Фактически, макросы являются альтернативой подпрограммам. Их преимущество в том, что тело подпрограммы с помощью макроса можно встроить непосредственно в программный код, т.е. не надо тратить время на вызов подпрограммы и на передачу параметров. Но т.к. макросы делают свое дело при компиляции, то они не могут проверять значения переменных, что накладывает большие ограничения на применение макросов (в частности, рекурсию с их помощью сделать нельзя). Кроме того, если подпрограмма большая, то затраты на ее вызов малы, а ее подстановка в программный код приведет к увеличению его объема.

Таким образом, макросы могут служить лишь заменой небольших подпрограмм. Но и в этом случае можно избавиться от них: написать компилятор так, чтобы он все небольшие подпрограммы встраивал в код или ввести специальную директиву (кстати, в С++ есть такая директива - inline), которая бы говорила компилятору, надо встраивать тело подпрограммы в код или нет. Быть может, для системного программирования макросы бывают иногда нужны, но в ОО-языке они явно лишние.

Базовые ОО-возможности С++

В С++, в отличие от Delphi, создаются сами объекты, а не ссылки на них. Поэтому если вы хотите создать объект в динамической памяти, то вы должны использовать указатели. Это приводит не только к тому, что смешиваются процедурный и ОО-стили. Возникают дополнительные вопросы, например, надо разграничивать понятия приведения объектов и приведения указателей на объекты.

В С++ основным понятием является класс. В С++ есть возможность ограничивать доступ к элементам класса. Для этого служат ключевые слова private, public, protected.

Private означает, что элемент класса доступен лишь внутри класса. Это разумнее, чем давать возможность доступа к private-элементу внутри модуля.

Есть в C++ статические функции (то же самое, что функции класса в Delphi) и статические переменные. В Delphi статических переменных нет, что, как мы уже говорили, не очень логично. С помощью статических переменных можно, например, контролировать количество созданных объектов данного класса. Без статических переменных сделать это проблематично.

В C++ есть одиночное наследование, динамическое связывание и полиморфизм.

Но в C++ нет базового класса, аналога TObject в Delphi. Это создает проблемы: наличие базового класса позволяет избавиться от необходимости использовать бестиповые указатели, т.к. ссылка на базовый класс уже дает необходимую общность. В результате для написания общих процедур приходится идти другим путем.

Расширенные возможности C++

Наследование (одиночное), инкапсуляция, полиморфизм есть в любом ОО-языке, начиная с Simula 67. Сейчас мы рассмотрим некоторые особенности ООП на C++. По необходимости мы ограничимся лишь констатацией фактов без подробных примеров и объяснений.

1. Множественное наследование

В C++ кроме обычного наследования есть наследование множественное. В таком случае классу-наследнику перейдут все поля и методы, которые объявлены как public и protected хотя бы в одном из классов-наследников. Возможность кажется заманчивой, но на самом деле проблем с множественным наследованием больше, чем реальной пользы от него.

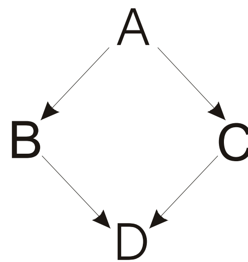


Рис. 20.1 Ромбовидное наследование

Разработчикам пришлось добавить механизм виртуальных базовых классов для того, чтобы избежать проблем ромбовидного наследования.

2. Перегрузка операций и операторов

В C++ можно перегружать операторы. Например, в C++ можно, проектируя класс для работы с дробями, переопределить операции +, - и др. для того, чтобы можно было их использовать и для дробей. Конечно, использование перегрузки операций часто делает программы более красивыми.

Но на мой взгляд, перегрузка операций излишня: она хорошо подходит для работы с классами, которые моделируют математические объекты, но в остальных случаях использование перегрузки операций может наоборот затуманить суть

программы. Кроме того, с перегрузкой тоже связаны трудности: чтобы она стала возможной разработчикам пришлось ввести понятие дружественных функций – функций другого класса, которые имеют право доступа к закрытым элементам данного класса. Ясно, что в таком случае теряется смысл понятия private, что нарушает инкапсуляцию.

3. Обобщенное программирование

Под обобщенным программированием понимают написание программ, которые могут использоваться не для конкретных, а для любых классов.

На процедурном уровне обобщенность достигается путем применения бестиповых указателей (когда мы с вами писали сортировку, которая упорядочивала что угодно и как угодно, мы занимались «обобщенным программированием»).

В Delphi обобщенность на уровне ООП достигается за счет трех фундаментальных идей:

1. Все классы – наследники TObject.
2. Для каждого объекта можно узнать его тип во время выполнения программы.
3. Есть метаклассы.

Эти идеи и просты, и красивы, и соответствуют реальному положению вещей, т.к. каждый объект содержит ссылку на класс. Больше ничего для достижения обобщенности не надо: можно передавать в методы ссылки на базовый класс, а конкретный тип объекта можно узнавать во время работы программы, т.к. есть встроенный метод ClassType, ClassParent. Можно передавать и ссылки на дескрипторы класса с помощью метаклассов, поэтому можно писать программный код, вообще абстрагируясь от конкретных объектов.

Но в С++ нет ни базового класса, ни метаклассов, - вместо этого для того, чтобы узнавать тип объекта, вводится несколько вспомогательных операторов. Для передачи классов вводятся шаблоны (это развитие идеи макросов).

Можно было бы говорить о С++ еще долго – в нем есть столько всего ненужного! – но мне кажется, что у вас уже и так сложилось некоторое впечатление о нем.

Java

Теперь поговорим о Java. Этот язык был разработан фирмой Sun Microsystems в 1996 году. Синтаксис языка взят из С, поэтому Java часто называют расширением языка С, иногда – «рафинированным С++». На самом деле по части ООП Java во многом похож на Delphi.

В Java, как и в Delphi:

- используются ссылки на объект, а не сами объекты
- есть базовый класс - Object.
- Наследование – только одиночное.
- Есть интерфейсы.
- Поддерживается механизм RTTI (Run Time Type Information), позволяющий определять тип объекта во время выполнения.

Метаклассов в Java нет, что не очень логично, т.к. из-за этого возможности Java по части обобщенного программирования существенно снижаются.

Java создавался как чистый ОО-язык, поэтому:

1. В Java вообще нельзя использовать указатели.
2. Макросов нет.
3. Нет оператора goto.

Однако в Java вводится неструктурный оператор, который позволяет выходить сразу из нескольких циклов (т.е. аналог break, но многоуровненый). Чтобы достичь общности, были введены шаблоны. Кроме того, синтаксис довольно путаный (все-таки Java – расширение С) и при этом он стал громоздкий (в отличие от С).

ОО-возможности Java примерно такие же, как и в Delphi, но отказ от средств, которые присущи процедурному программированию, делает Java языком с ограниченной областью применения.

19.3. Компиляция или интерпретация

Программы, которые переводят программный код на машинный язык бывают двух типов: компиляторы, переводящие сразу весь программный код на машинный язык и интерпретаторы, которые переводят оператор языка высокого уровня в машинные инструкции и выполняют его, потом переходят к следующему оператору.

Если программный код будет компилироваться, то мы получим готовую программу на машинном языке некоторого процессора. Эта программа может быть выполнена на любом компьютере с этим процессором и ей уже не нужен ни исходный код, ни сам компилятор. Но если процессор на ПК другой, то эта программа не сможет быть выполнена на этом процессоре (точнее, это вообще не будет программой для данного процессора, т.к. мы определили понятие программы как последовательности команд, которая может быть выполнена на процессоре).

Можно было бы решить проблему так: передавать исходный код (быть может, некоторым образом закодированный, чтобы его никто не смог прочитать), а затем компилировать проект на той машине, на которой будет выполняться проект. Но компиляция больших проектов занимает много времени, поэтому такой выход – не самый лучший.

Если для перевода программного кода в машинные инструкции используется интерпретатор, то он должен быть установлен на машине, на которой должен выполняться проект. Тогда можно передавать исходный код на эту машину, после чего он будет оператор за оператором выполняться с помощью интерпретатора. Скорость работы программы будет меньше, чем при использовании скомпилированного программного кода, но зато исходный код может быть выполнен на компьютере с любым процессором (но для каждого типа компьютера нужен свой интерпретатор).

Для того чтобы оставить механизм интерпретации, но при этом достичь большей скорости выполнения программ, используются комбинированные механизмы.

В качестве примера рассмотрим JVM (Java Virtual Machine).

Исходный код на языке Java компилируется в программу на байт-коде – промежуточном низкоуровневом языке. Выполняется этот программный код с помощью JVM – интерпретатора байт-кода. Для того чтобы программы, написанные на Java, могли выполняться на компьютере, на нем должна быть установлена JVM.

Конечно, не следует считать, что программы на Java должны обязательно преобразовываться в байт-код, а потом интерпретироваться – в принципе можно было

бы написать и компилятор для языка Java. Просто фирма Sun, разработавшая язык Java изначально делала упор на интерпретируемость языка.

19.4. Недостатки объектно-ориентированных языков

Описание самоорганизующихся систем

Некогда вы были обезьяной, и даже теперь еще
человек больше обезьяны, чем иная из обезьян
Фридрих Ницше «Так говорил Заратустра»

Основное преимущество ООП перед модульным программированием в том, что программист может мыслить в терминах классов и объектов, что гораздо ближе к реальности, чем мыслить модулями, которые представляются просто как коллекции подпрограмм. Более того, принципы ООП могут реализовываться на основе логических и функциональных языков, таких как Prolog и Lisp.

Но ООП – не панацея от всех бед и слова Ницше из эпиграфа к этому пункту могут быть смело применены к ООП. То, что объекты могут беспрепятственно вызывать методы друг друга, зачастую противоречит тому, что мы видим в реальной жизни.

Классы более-менее справляются, когда надо моделировать неживые объекты, но при описании самоорганизующихся систем возникают проблемы. Система, состоящая из классов похожа на механизм, состоящий из подмеханизмов, каждый из которых может непосредственно влиять на другие. Но любой объект с внутренней организацией обладает определенной самостоятельностью, и не дает возможности вызывать свои функции извне – он может получать информацию, затем обрабатывать ее как ему вздумается и, если ему хочется, возвращать кому-то ответ. Заставить его что-то обязательно отдать и, тем более, полностью руководить им, извне нельзя. На программистском языке можно сказать так: **у существа вообще нет открытых полей и методов**. Более того, организм не может получить полный доступ даже к своим собственным составным частям: я могу двигать ногами и головой, дышать носом, ощущать определенные изменения моего самочувствия, но не способен влиять на биохимические процессы в клетках. **Существо может влиять на свое внутреннее состояние, но непосредственного контроля над каждым внутренним процессом у него нет.**

Объектно-ориентированное программирование не дает практически никаких средств описания систем с самоорганизацией. Единственный, и притом очень скромный шаг в этом направлении – введение свойств: я могу что-то взять или передать объекту, но при этом включаются его внутренние механизмы, которые реагируют на это.

Объекты языкового уровня и объекты времени выполнения

Когда программист пишет программу, он использует объекты языкового уровня. Программист, исповедующий принципы ООП, должен стараться как можно лучше структурировать программный код, разбив его на описания ряда классов. Но императивные языки (и их ОО-расширения в том числе) сохранили понятие главной

программы, которая представляет собой просто последовательность операторов. Конечно, теперь в ней вызываются методы глобальных объектов, однако основная программа все равно представляет собой не главный класс – а лишь **основную подпрограмму**, которая оперирует с объектами. Более того, программист то и дело **должен использовать функции ОС**, которые не принадлежат ни к каким объектам. То, что в Delphi есть класс TApplication, который берет на себя добрую часть взаимодействия с ОС, и целая библиотека встроенных классов в придачу, не должно вводить нас в заблуждение относительно их содержимого – все они должны использовать функции ОС.

Мы обсудили программный код, теперь давайте рассмотрим программу во время выполнения. Как только программа скомпилирована и запущена на выполнение, она понимается ОС не как набор классов, а как процесс, который состоит из ряда потоков (в Windows потоки делятся еще и на волокна), у которого есть свое адресное пространство, сегмент стека и т.д. Короче, появляется целый ворох понятий, относящихся непосредственно к жизненному циклу программы.

- Объекты времени выполнения (ОВВ) – понятия, описывающие работу программы.

  В частности, процесс, поток, волокно, - ОВВ.

Взаимодействие программы с операционной системой

В главе 14 мы определили понятие операционной системы так:

- ОС – программа, являющаяся прослойкой между аппаратным обеспечением и прикладными программами и предоставляющая функции, позволяющие прикладным программам работать с аппаратным обеспечением и взаимодействовать друг с другом.

Т.е. ОС – это нечто большее, чем обычная прикладная программа.

Механизм сообщений, многие объекты времени выполнения, дополнительные понятия, которые служат лишь для поддержки «новых технологий», сильно загромождают язык. Все это говорит лишь о том, что должны меняться принципы программирования, которые не развивались со времен Simula и Smalltalk.

Часто ставится вопрос о возможно большей независимости программ от ОС. Но уже сама его постановка говорит о том, что ОС является чем-то отличным от обычной прикладной программы. В рамках существующих концепций программирования не может быть дано удовлетворительное определение того, что такое ОС – это понятие всегда оставалось внешним.

19.5. Несколько терминов из биологии

Объектно-ориентированное программирование родилось (язык Simula 67) из благого стремления людей исследовать мир, поэтому не удивительно, что многие термины ООП пришли из биологии. Даже понятие функции стало пониматься не в математическом (как в процедурном программировании), а в биологическом смысле:

- Функция – специфическая деятельность животного или растительного организма, его органов, тканей и клеток.
- Полиморфизм –

1. способность некоторых кристаллических веществ образовывать в зависимости от условий несколько различных по кристаллической структуре и другим физическим свойствам модификаций без изменения состава вещества (графит – алмаз).
2. Наличие в пределах одного того же вида животных или растений особей, резко отличающихся друг от друга; различают половой, возрастной, сезонный и связанный с чередованием поколений; у растений полиморфизмом называют также наличие у одной особи различных форм какого-либо органа.

Мы продолжим традицию использования биотерминов, и вспомним еще несколько:

- Генетический код – запись наследственной информации в виде последовательности кодонов.
- Кодон – единица генетического кода, состоящая из трех азотистых оснований (нуклеотидов). Кодон предписывает включение определенной аминокислоты в синтезируемую молекулу белка.
- Фенотип – совокупность всех признаков и свойств организма, сформировавшихся в процессе его индивидуального развития.

С точки зрения программиста, генетический код существа – программа, кодон – элементарная команда (она анализируется информационной РНК).

Нам понадобятся эти понятия, когда мы введем понятие существа (Wesen).

19.6 Везенспрограммирование (Wesensprogrammierung)

Основные понятия

Основным понятием ООП было понятие класса. Инкапсуляция реализована на уровне классов, а не объектов: объект класса А может изменять в своих методах значения private-полей любого другого объекта класса А.

Объект – производное понятие: объект - экземпляр класса.

Базовым понятием везенспрограммирования (ВП) будет понятие существа (Wesen). Существо – аналог объекта. Его описание (аналог класса) – назовем Metawesen.

Wesen состоит из трех частей:

1. Hirn (мозг)
2. Inwelt (внутренний мир)
3. Zutritte (входы)

Каждое существо живет в своем собственном потоке.

Metawesen описывается так:

```
type
  MW = Metawesen
    Hirn:THirn
    Inwelt:
      …
  end;
```

- **Hirn** – это объект, своеобразный центр управления у существа.

- **Inwelt** – это внутренний мир существа. Внутренний мир существа может состоять как из объектов, так и из других существ.
- **Zutritt** (вход) – единственное средство, с помощью которого существо может принимать информацию от других существ. Объекты ничего не могут посылать существам, они - лишь источник данных (как в ООП).

Для того чтобы посылать информацию другим существам, вводится понятие вызова (**Ruf**). Чтобы принимать информацию, существо должно включить входы («настроиться на волну»). Если существо не хочет принимать информацию извне, то входы блокируются.

Информация, которую получает существо, может храниться внутри Hirn в какой-то структуре данных. По мере надобности Wesen может анализировать полученную информацию.

У каждого существа есть Inwelt, в котором могут жить другие существа, к которым существо, в котором они живут, может обращаться лишь при помощи вызовов. С другой стороны, каждое существо тоже обитает в Inwelt какого-то другого существа.

- Если существо А находится в Inwelt существа B, то B называется надсуществом (**Oberwesen**) существа А.
- Inwelt надсущества для существ, которые в нем обитают, будет называться **Umwelt** (окружающий мир).

Если бы каждое существо должно было бы обитать в некотором надсуществе, то получилась бы бесконечная последовательность надсуществ. Этого быть не может, поэтому есть главное существо (**Hauptwesen, HW**), которое не входит в Inwelt какого бы то ни было надсущества. Это главное надсущество – как вы уже догадались, и есть то, что в старину называли ОС.

Заметьте, что понятие главного существа не является для везенспрограммирования внешним, и при этом оно полностью соответствует старому понятию ОС:

- Все существа, обитающие в инвельте главного существа не могут непосредственно получить доступ к аппаратуре, следовательно, главное существо скрывает аппаратуру и предоставляет возможность обращаться к ней, используя его функции.
- Существа, обитающие в инвельте главного существа могут общаться друг с другом (через главного существа).
- Т.к. каждое существо живет в собственном потоке, то выходит, что главное существо распределяет ресурсы между ними.

Таким образом, существование главного существа является необходимым следствием концепции везенспрограммирования, а не каким-то внешним понятием, взятым непонятно откуда.

Жизнь существа

Итак, каждое существо обитает в окружающем пространстве (Umwelt). Кроме того, у каждого существа есть внутреннее пространство (Inwelt).

Во внутреннем пространстве существо может обращаться к любому из объектов, а к существам, которые в нем живут, может обращаться при помощи вызовов. Существо может создавать своих подсуществ и уничтожать их.

В Umwelt права существа более ограничены: обратиться к другим существам оно может с помощью вызовов к ним. А обращение к объектам Umwelt можно проводить через вызов к надсуществу, которое руководит «мертвой материей» в своем Inwelt.

Вас не должна смущать некоторая громоздкость обращения к внешним объектам. Во-первых, очень маловероятно, что на уровне существ будут встречаться целые числа или какие-то простые объекты. Наверняка объектами, с которыми работает существо, являются какие-то базы данных, файлы и т.д. Когда мы в программах обращались к файлу, то мы в конечном счете использовали функции ОС, поэтому обращение к надсуществу – это просто обобщение тех механизмов, которые и так используются.

В то же время ввод посредника позволяет сделать существо более независимым от окружения. Мы еще обсудим этот тонкий момент позже, а пока что попрошу вас заметить, что мы уже говорим о существах, как о чем-то живом, т.к. они существуют в своем потоке и живут своей жизнью.

Подытожим, чего мы уже добились:

- обособленности (выполнение в собственном потоке)
- самостоятельности (независимость от других существ и внешних объектов)
- полной инкапсуляции (повлиять на существо можно лишь с помощью вызовов и то если оно само того хочет).

Давайте посмотрим, как ВП избавляет программиста от ненужных терминов.

Убираем ненужные технические понятия

Когда программист пишет программу согласно традиционной методологии программирования (т.е. не используя ВП), он использует множество понятий, которым нет прямых аналогов в природе: процесс, поток, программа, операционная система. Совершенно очевидно, что значения слов «процесс» и «поток» в программировании и обиходной речи различаются.

В ВП ситуация иная: существо (Wesen) живет, обмениваясь информацией с окружающей средой (Umwelt), которую обрабатывает мозг (Hirn) этого существа. Все эти понятия в ВП полностью соответствуют понятиям из реальной жизни, поэтому они гораздо яснее, чем притянутые за уши технические термины. Даже понятие программы теперь может быть выражено так:

- программа - генетический код существа.
- генетический код – последовательность выполняемых неделимых команд.

Для живых существ код выполняется в клетке, а для информационных – в процессоре.

Существо ссылается на дескриптор Metawesen, который ссылается на методы, которые есть у всех существ этого Metawesen. Программный код, описывающий Metawesen – это описание генетического кода существа.

Сам «генетический код» существа – это последовательность команд, которая отводится под само существо, вместе с дескриптором Metawesen и описанием мозга (Hirn). Заметьте, что даже понятие программы – последовательности команд в ВП заменяется «генетическим кодом существа».

Фенотипом существа будет генотип + переменные, которые использует существо во время работы (дополнительная память).

В таком случае процесс – это фенотип существа, обитающего в ОС.

Давайте приведем небольшую таблицу, в которой сравним некоторые старые и новые понятия:

Старые понятия	Новые понятия
1. программа	генотип существа
2. процесс	фенотип существа
3. поток	не используется
4. Операционная система (ОС)	Hauptwesen (HW)
5. написать программу	описать существо
6. запустить программу	добавить существо в среду
7. сообщения	вызовы

Понятие потока в явном виде вообще не используется, т.к. поток неразрывно связан с существом. Конечно, иногда существу нечего делать, то надо дать ему возможность «впадать в спячку». Выход из нее можно реализовать по-разному: например, как только будет накоплено достаточное количество входных данных, то он может проснуться и обработать входную информацию.

Читатель может возразить что, принимая везенспрограммирование, мы хотя и избавляем программиста от старых понятий, но при этом вводим взамен новые. Но на самом деле мы не вводим новые термины, а лишь расширяем значение уже давно известных биологических понятий на информационный мир, что естественнее и, следовательно, правильнее.

Описание существ на разных языках

При включении компьютера автоматически рождается существо – Hauptwesen. Его Inwelt может быть изначально пустым. Во время работы оно (Hauptwesen) может добавлять в свой Inwelt другие существа (по собственной инициативе или по приказу пользователя). Каждое из этих существ также может впоследствии пополнять свой Inwelt различными существами.

Программист, если хочет описать новое существо (раньше мы бы сказали: написать новую программу), пишет Metawesen. После его компиляции создается файл с кодом, описывающим Metawesen. Операция добавления нового существа в Inwelt какого-то другого существа заключается в том, что должен быть запущен метод, который создает существо (назовем его Leben - жизнь), а код Metawesen подключится в Inwelt существа.

Т.к. существа связаны с внешним миром (и с внутренними существами) лишь посредством вызовов, то для того, чтобы можно было описывать существа на разных языках, надо лишь:

1. Общий механизм обработки вызовов
2. Общий механизм добавления существ в среду.

При выполнении этих условий существа можно будет описывать на любых языках программирования.

Оба эти условия нужны и для традиционного подхода к выполнению программ (они проявляются в том, что все программы должны быть в состоянии выполняться под управлением ОС).

Общая картина

На каждом компьютере есть Hauptwesen – главное существо. Компьютеры, объединенные в сеть, образуют пространство, в котором живут существа, которые могут обмениваться информацией друг с другом. Глобальная сеть представляет собой информационное пространство (Inforaum), в которой живут существа, которые могут обмениваться информацией друг с другом и с существами, которые живут в их Inwelt. Информационная материя бывает живая и мертвая. Живая представлена существами, а мертвая – объектами.

Если посмотреть на программирование с точки зрения ВП, то можно пересмотреть определение программирования, данное нами в 0-й главе (программирование – это написание программ). Теперь задача программиста – не давать указания процессору, что ему делать, а описывать явления, происходящие в Inforaum.

Видно, что концепция везенспрограммирования решает те 3 проблемы, которые были рассмотрены выше.

19.7. Что мы с вами еще не сделали

Мы не рассмотрели еще один недостаток современных языков программирования – ограниченность наследования. Наследование позволяет создавать новые классы на основе уже существующих. Но для этого предлагается лишь один способ: дописать что-то или изменить что-то (т.е. переопределить метод). Исходя из этого, мне кажется, что наследование следовало бы назвать *обрастанием*.

Кроме того, одиночного наследования не всегда хватает, а с реализацией множественного наследования могут возникнуть проблемы. А хотелось бы получить универсальный механизм, который не ограничивался бы добавлением или изменением каких-то частей класса, но позволял бы перестраивать класс как угодно, причем чтобы большую часть этой работы делал бы сам компьютер, а не программист. Пока такого механизма нет. Но зато есть задача, которая требует решения. Так что вперед!

Напутствие

Учебник подходит к концу. На мой взгляд, у вас может возникнуть два вопроса:
Чего мы достигли?
Что делать дальше?

Первый вопрос значительно проще второго. Мы с вами:

- Изучили высокоуровневые императивные языки программирования: рассмотрели общие принципы, которые положены в основу процедурных и ОО-языков и как эти принципы реализованы в TP и Delphi, а также вкратце обсудили некоторые другие языки.
- Рассмотрели многие важные алгоритмы и основные структуры данных и разобрались, зачем они вообще нужны.
- Изучили некоторые принципы работы ОС и разобрали проблемы, которые возникают при взаимодействии ОС и концепций ООП.
- Разработали концепцию везенспрограммирования (ВП), которая позволяет решить многие из проблем, с которыми столкнулась современные ОО-языки.

Второй вопрос гораздо важнее.
На мой взгляд, дальше можно:

1. Изучить функциональные (напр. LISP) и логические (напр. Prolog) языки. Это не отнимет у вас много времени, т.к. часть понятий в этих языках будет аналогична тем, которые вам уже известны, и вам не надо заново изучать алгоритмы (не говоря о том, что ваш опыт программирования уже внушителен).
2. Изучить низкоуровневое программирование. Сложность ассемблера – чисто техническая, концептуально он очень прост. Поэтому начинать изучение с ассемблера не очень эффективно. Но теперь, когда вы уже хорошо понимаете программирование, разобраться в том, как функционирует процессор, будет очень полезно. Кроме того, интересно знать, как реализуется рекурсия на низком уровне, насколько быстрее можно выполнить умножение, чем сложение и т.д. Эти вопросы удобнее всего рассматривать именно в курсе ассемблера.
3. Полезно разобраться в устройстве современных ОС и компиляторов.
4. Искусственный интеллект (ИИ). Несмотря на многочисленные высказывания о том, что вот-вот будет построен машинный интеллект, который будет соперничать с человеческим, все это на сегодняшний момент – лишь разговоры. Но имитация (довольно грубая) некоторых аспектов умственной деятельности проводится уже давно. Хотя это и не ИИ, но все же эти алгоритмы позволяют решать достаточно сложные задачи. Поэтому я думаю, что разобраться в современных тенденциях в исследовании ИИ и отделить зерна от плевел будет для вас интересной задачей.

На последние вопросы я ответил – теперь можно прощаться. Надеюсь, что этот учебник вам понравился и еще больше – что кто-то из вас напишет другой, который будет лучше этого. Удачи вам!

Решения и ответы

Глава 1

1. Такого числа нет.
3. $1094_2 = 100\ 0100\ 0110$, $1024_2 = 100\ 0000\ 0000$
4. $BAD_{16} = 2989_{10}$, $FEED_{16} = 65261_{10}$
5. $s = [\log_{10} n] + 1$
6. $s = [\log_k n] + 1$, k - основание системы счисления.
7. Пусть $x = 0.123456789101112...$ Предположим, что x - периодическая дробь. Тогда оно представимо в виде: $x = y + 0.(T)$, где y - конечная десятичная дробь, а $T = t_1t_2...t_n$ - период (t_i - цифры). Берем первое из чисел натурального ряда, находящихся в периодической части числа x такое, что его количество цифр будет кратно периоду (ясно, что такое число существует). Пусть это число = s. Вид его будет таким: $s = t_i...t_nt_1...t_n...t_1...t_{i-1}$. Но в таком случае следующее число должно совпасть с числом s, т.к. у него столько же цифр, сколько и у числа s (если s состоит из одних девяток, тогда период состоит из девяток, что неверно). Но этого не может быть, т.к. следующее число на 1 больше, чем s. Получили противоречие.
8. Согласно ПИ $a^{k-1} = 1$, $k = \overline{1,n}$. Подставляя в выкладки, приведенные в «доказательстве» значение $n = 1$, получим, что $a^{(1+1)-1} = \frac{a^{1-1} \cdot a^{1-1}}{a^{(1-1)-1}}$, т.е. в знаменателе стоит a^{-1}, а в ПИ не говорится о том, что $a^{-1} = 1$, следовательно, ПИ применять нельзя.
9. Ошибка в БИ. Сумма, стоящая в левой части, будет равна $\frac{1}{1 \cdot 2}$ при $n = 2$. При $n = 1$ она равна 0.
10.

БИ: при $n = 1$ неравенство очевидно

ПИ: предположим, что для любого $k \le n$ выполняется $(1-a_1)(1-a_2)...(1-a_k) \ge 1-(a_1+a_2+...+a_k)$

ШИ: докажем, что неравенство выполняется и для $k = n+1$:

$$(1-a_1)(1-a_2)...(1-a_n)(1-a_{n+1}) \ge [\text{согласно ПИ и тому, что } a_i < 1] \ge (1-(a_1+a_2+...+a_n))(1-a_{n+1}) =$$

$$= 1-(a_1+a_2+...+a_{n+1}) + a_{n+1}(a_1+...+a_n) \ge [\text{т.к. } a_i > 0, i = \overline{1,n}] \ge 1-(a_1+a_2+...+a_{n+1})$$

14. От противного: $\log_3 15 = \frac{m}{n} \Rightarrow 3^{\frac{m}{n}} = 15 \Rightarrow 3^m = 3^n5^n$. Если $n \ne 0$, то правая часть делится на 5, а левая нет. Значит $n = 0$, а этого не может быть. Получили противоречие.
15. Пусть q_k - количество k-ичных цифр в записи числа s_k. Так как $f_k = 0.(s_k)$, то $(10_k)^{q_k} f_k = s_k.(s_k)$, где 10_k - число k, записанное в k-ичной ССч. Полученное выражение можно преобразовать так: $(10_k)^{q_k} f_k = s_k + f_k$. Осталось лишь перевести все числа из k-ичной ССч в 10-ю: $k^{q_k} f_{10} = s_{10} + f_{10}$, или $f_{10} = \frac{s_{10}}{k^{q_k} - 1}$.
18. Докажем соотношение. Шаг индукции:

$$y^{(n+1)}=(y^{(n)})'=\left((-1)^n n!\,x^{-n-1}\ln x+(-1)^{n-1}c_n x^{-n-1}\right)'=(-1)^{n+1}(n+1)!\,x^{-n-2}\ln x+(-1)^n((n+1)c_n+n!)x^{-n-2}$$

Если принять, что $c_{n+1}=(n+1)c_n+n!$, то соотношение доказано. Теперь найдем вид c_n. Давайте вместо c_n подставим ее выражение через c_{n-1}.

$c_{n+1}=(n+1)\left((n-1)!+c_{n-1}n\right)+n!=n!+(n-1)!(n+1)+c_{n-1}n(n+1)$.

Теперь пришло время догадки. Видно, что $n!=\frac{(n+1)!}{n+1}$, $(n-1)!(n+1)=\frac{(n+1)!}{n}$. Если внимательно приглядеться, то сразу приходит в голову мысль: $c_n=n!\left(1+\frac{1}{2}+\frac{1}{3}+...+\frac{1}{n}\right)$.

Осталось лишь доказать это соотношение по индукции.

19. а) $\varphi=\lim\limits_{n\to\infty}\frac{f_{n+1}}{f_n}=\lim\limits_{n\to\infty}\frac{f_n+f_{n-1}}{f_n}=1+\lim\limits_{n\to\infty}\frac{f_{n-1}}{f_n}=1+\frac{1}{\varphi}$. Значит $\varphi^2-\varphi-1=0$. Отсюда, учитывая, что $\varphi>0$, получим: $\varphi=\frac{1+\sqrt{5}}{2}$ - «золотое сечение»

б) Мы знаем: $\varphi=1+\frac{1}{\varphi}$. Оценим разность $\frac{f_{n+1}}{f_n}-\varphi$.

$$\frac{f_{n+1}}{f_n}-\varphi=\frac{1}{f_n}(f_{n+1}-\varphi f_n)=\frac{1}{f_n}\left(f_{n+1}-(1+\frac{1}{\varphi})f_n\right)=\frac{1}{f_n}\left(f_{n-1}-\frac{1}{\varphi}f_n\right)=\frac{-1}{\varphi f_n}\left(f_n-\varphi f_{n-1}\right)=[...]=$$

$$=\frac{(-1)^2}{\varphi^2 f_n}\left(f_{n-1}-\varphi f_{n-2}\right)=[...]=\frac{(-1)^n}{\varphi^n f_n}(f_1-\varphi f_0)=\frac{(-1)^n(1-\varphi)}{\varphi^n f_n}=\frac{A_n}{\varphi^n}$$

$$\left|\frac{f_{n+1}}{f_n}-\varphi\right|=\frac{|A_n|}{\varphi^n}\le\left[\forall n\in N\Rightarrow |A_n|<2\right]\le\frac{2}{\varphi^n}\xrightarrow[n\to\infty]{}0.$$

20. Мы знаем (задача 19), что $\varphi=1+\frac{1}{\varphi}$. Сразу хочется сделать так:

$\varphi=1+\cfrac{1}{1+\cfrac{1}{\varphi}}=1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{\varphi}}}=...=1+\cfrac{1}{1+\cfrac{1}{1+...}}$. Идея верна, однако надо доказать сходимость правой части равенства к золотому сечению.

Давайте введем такую последовательность функций: $g_1=1+\frac{1}{x}$, $g_{n+1}=1+\frac{1}{g_n}$, $n>1$.

Пусть $g_n=\frac{A_n}{B_n}$ - несократимая дробь, где A_n, B_n - многочлены от переменной x. Тогда $\frac{A_n}{B_n}=g_n=1+\frac{1}{g_{n-1}}=1+\frac{B_{n-1}}{A_{n-1}}=\frac{A_{n-1}+B_{n-1}}{A_{n-1}}$, т.е. $B_n=A_{n-1}$, и $A_n=A_{n-1}+A_{n-2}$. Получились какие-то многочлены Фибоначчи. Из $g_1=\frac{1+x}{x}$, $g_2=\frac{1+2x}{x+1}$ можно предположить: $g_n=\frac{f_n x+f_{n-1}}{f_{n-1}x+f_{n-2}}$, что легко доказывается по индукции. Пусть φ - золотое сечение.

$$\lim_{n\to\infty}g_n=\lim_{n\to\infty}\frac{f_n x+f_{n-1}}{f_{n-1}x+f_{n-2}}=\lim_{n\to\infty}\frac{(f_n/f_{n-1})x+1}{x+(f_{n-2}/f_{n-1})}=\frac{\varphi x+1}{x+(1/\varphi)}=\varphi\frac{x+(1/\varphi)}{x+(1/\varphi)}=\varphi$$

21. Рассмотрим $S_{p+1}(n+1)=1^{p+1}+2^{p+1}+...+(n+1)^{p+1}=1^{p+1}+(1+1)^{p+1}+(2+1)^{p+1}+...+(n+1)^{p+1}=$.

$$=1^{p+1}+\sum_{i=0}^{p+1}C_{p+1}^{i}1^{i}\cdot 1^{p+1-i}+\sum_{i=0}^{p+1}C_{p+1}^{i}2^{i}\cdot 1^{p+1-i}+...+\sum_{i=0}^{p+1}C_{p+1}^{i}n^{i}\cdot 1^{p+1-i}=1+\sum_{i=0}^{p+1}C_{p+1}^{i}(1^{i}+2^{i}+...+n^{i})=$$

$$=1+\sum_{i=0}^{p+1}C_{p+1}^{i}S_{i}(n)=1+S_{p+1}(n)+(p+1)S_{p}(n)+\sum_{i=0}^{p-1}C_{p+1}^{i}S_{i}(n).$$

Теперь легко видеть, что $(n+1)^{p+1}=1+(p+1)S_{p}(n)+\sum_{i=0}^{p-1}C_{p+1}^{i}S_{i}(n)$.

Следовательно: $S_{p}(n)=\frac{1}{p+1}\left((n+1)^{p+1}-1-\sum_{i=0}^{p-1}C_{p+1}^{i}S_{i}(n)\right)$.

Рекуррентная формула выведена. Надо еще знать $S_0(n)$: $S_0(n)=1^0+2^0+...+n^0=n$.

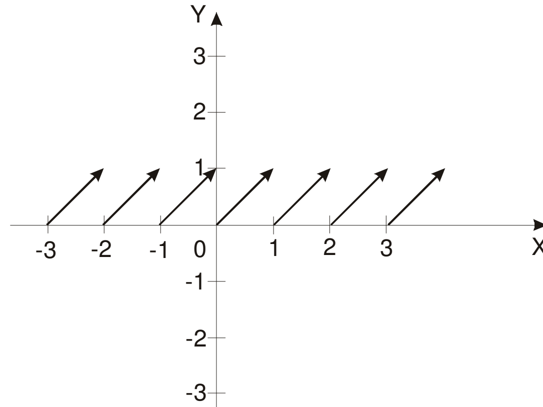
Глава 2

5. Объем бочки: $V_{бочки}=\pi(R-c)^2(H-2c)$, где c – толщина.

   Масса пустой бочки: $M_{бочки}=V\cdot\rho_{жел}=(\pi R^2H-V_{бочки})\cdot\rho_{жел}$

   V - объем железа, затраченного на изготовление бочки.

6.



7. Предположим, что T – период, тогда $\{x+T\}=\{x\}=x-[x]$

Но при этом $\{x+T\}=x+T-[x+T]$.

Объединяя результаты, и приводя подобные слагаемые, получим:

$$[x]+T=[x+T] \qquad (1)$$

Это равенство должно выполняться при всех x, в частности, для x=0. Значит $T=[T]$, а это выполняется только для целых чисел. А в том, что формула (1) выполняется для целых чисел T, можно убедиться непосредственно из определения функции целой части. Значит функция $y=\{x\}$ периодическая, и ее период равен 1.

8. Равенство верно лишь для дробных значений x. Для целых чисел равенство не выполняется.

9. Используя периодичность функции $y=\{x\}$, получим:

$\{x+y\}=\{\{x\}+[x]+\{y\}+[y]\}=\{\{x\}+\{y\}\}$

Теперь, учитывая то, что $\{x\}\le x$ при любых положительных числах x, получим, что

$\{\{x\}+\{y\}\}\le\{x\}+\{y\}$, что и доказывает утверждение.

10. $[x+y]=x+y-\{x+y\}\ge x+y-(\{x\}+\{y\})=[x]+[y]$

11. Будем доказывать, что из $f_1(x+y)\le f_1(x)+f_1(y)$ следует $f_2(x+y)\ge f_2(x)+f_2(y)$.

$f(x+y)=f_1(x+y)+f_2(x+y)\le f_1(x)+f_1(y)+f_2(x+y)$

Из аддитивности $f(x)$ следует $f(x+y)=f(x)+f(y)=f_1(x)+f_2(x)+f_1(y)+f_2(y)$. Подставив это равенство в предыдущее неравенство и приводя подобные слагаемые, получим требуемое утверждение.

Аналогичные выкладки можно провести и в обратную сторону (тем, кто не сумел справиться с упражнением прежде, чем заглянуть в ответы, рекомендуется эти выкладки провести).

Очевидно, что результат задачи 10 – частный случай доказанного в этом упражнении утверждения.

12. $$\min\left(d\left(\frac{m}{6n}\right),\ d\left(\frac{m}{3n}\right)\right)=\begin{cases}\min(\frac{m}{6n},\ \frac{m}{3n}), & \frac{m}{3n}\le\frac{1}{2}\\ \min(\frac{m}{6n},\ 1-\frac{m}{3n}), & \frac{1}{2}\le\frac{m}{3n}\le 1\\ \min(1-\frac{m}{6n},\ \frac{m}{3n}-1), & 1\le\frac{m}{3n}\le\frac{3}{2}\\ \min(1-\frac{m}{6n},\ 2-\frac{m}{3n}), & \frac{3}{2}\le\frac{m}{3n}\le 2\end{cases}=\begin{cases}\frac{m}{6n}, & m\le 2n\\ 1-\frac{m}{3n}, & 2n+1\le m\le 3n\\ \frac{m}{3n}-1, & 3n+1\le m\le 4n\\ 1-\frac{m}{6n}, & 4n+1\le m\le 6n\end{cases}$$

Первое равенство следует непосредственно из определения функции $d(x)$, а второй переход получен непосредственно рассмотрением каждого из членов системы, например:

$$1-\frac{m}{3n}-\frac{m}{6n}\le 0\Leftrightarrow 1-\frac{m}{2n}\le 0\Leftrightarrow m\ge 2n,$$

Значит: $\min(\frac{m}{6n},\ 1-\frac{m}{3n}),\quad \frac{1}{2}\le\frac{m}{3n}\le 1$ эквивалентно $\min(\frac{m}{6n},\ 1-\frac{m}{3n})=\begin{cases}\frac{m}{6n}, \frac{3n}{2}\le m\le 2n\\ 1-\frac{m}{3n}, 2n\le m\le 3n\end{cases}$

Следовательно, $F_n=\sum_{m=1}^{2n}\frac{m}{6n}+\sum_{m=2n+1}^{3n}\left(1-\frac{m}{3n}\right)+\sum_{m=3n+1}^{4n}\left(\frac{m}{3n}-1\right)+\sum_{m=4n+1}^{6n}\left(1-\frac{m}{6n}\right)=[...]=n$.

Глава 3

3. Основная сложность этой задачи в том, как перераспределить значения. Существует несколько способов. Один из них:

```
tmp:=x;
x:=y;
y:=tmp;
```

Здесь tmp – вспомогательная переменная.

Переприсваивание можно сделать и без использования дополнительных переменных:

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

Или так:

```
x:= x xor y;
y:= x xor y;
x:= x xor y;
```

Однако лучше использовать первый вариант, т.к. во втором и третьем методе кроме операций присваивания производятся дополнительные действия, на что

затрачиваются ресурсы процессора, и программа будет работать медленнее, особенно, если операция переприсваивания должна проводиться большое число раз (например, при сортировке массивов методом пузырька (см. главу «Массивы»)).

8. Сумма цифр трехзначного числа x = (x div 100) + ((x mod 100) div 10)+(x mod 10).

10. Для доказательства можно воспользоваться, например, методом выделения квадрата двучлена. Преобразуем уравнение $ax^2+bx+c=0$:

Сначала разделим все уравнение на a, затем прибавим и отнимем $\frac{b^2}{4a^2}$.

$$\left(x^2+\frac{b}{a}x+\frac{b^2}{4a^2}\right)-\frac{b^2}{4a^2}+\frac{c}{a}=0$$

В скобках выделим квадрат двучлена, а остальные члены уравнения перенесем в правую часть.

$$\left(x+\frac{b}{2a}\right)^2=\frac{b^2-4ac}{4a^2}$$

Если $D=b^2-4ac<0$, то уравнение действительных корней не имеет. В противном случае извлечем квадратный корень.

$$x+\frac{b}{2a}=\pm\frac{\sqrt{D}}{2a},\ D=b^2-4ac$$

Итого: $x=\frac{-b\pm\sqrt{D}}{2a}$, если $D\ge 0$. Если $D<0$, то действительных корней нет.

14. $(x^x)'=(e^{x\ln x})'=e^{x\ln x}(1+\ln x)=x^x(1+\ln x)$

Можно обобщить пример, и искать производную $\left((f(x))^{g(x)}\right)'$.

Ответ: $\left((f(x))^{g(x)}\right)'=(f(x))^{g(x)}\left(g'(x)\ln f(x)+\frac{f'(x)g(x)}{f(x)}\right)$

16. x or y = not(not(x) and not(y))
17. x and y = not(not(x) or not(y))
18. x xor y = (x or y) and (not (x and y))

Глава 4

6. Есть много способов решения этой задачи. Давайте рассмотрим некоторые из них (до которых я додумался)

1 вариант – в цикле каждый раз вычислять значения x^k, $k=\overline{1,10}$, умножать это значение на соответствующий коэффициент при степени, и прибавлять каждое слагаемое к общей сумме. При этом степень считать по формуле $x^k=e^{k\ln(x)}$. Реализация этого метода:

```
S:=11;
for i:=10 downto 1 do
  S:=S+(11-i)*exp(i*ln(x));
```

Но вычисление логарифма и экспоненты от числа – очень трудоемкие операции, поэтому желательно найти более эффективный алгоритм.

2 вариант решения: если суммировать не с начала, а с конца, то легко заметить, что степени одночленов будут увеличиваться на 1. Это можно выгодно использовать: завести вспомогательную переменную, в которой будет храниться соответствующая степень переменной x. Увеличивая значения этой переменной с каждым витком цикла

на x, мы избавимся от необходимости использования функции вычисления степени числа.

```
S:=11;
y:=1;
for i:=10 downto 1 do
  begin
  y:=y*x;
  S:=S+i*y;
  end;
```

При такой реализации алгоритма мы затратим лишь 20 умножений и 10 сложений.

3 вариант решения еще лучше: давайте преобразуем выражение следующим образом:

$x^{10}+2x^9+...+10x+11=((((x+2)x+3)x+4)x+...+10)x+11;$

Это – так называемая схема Горнера. Сначала будем вычислять значение скобки с наибольшей степенью вложенности, и умножать ее на x. Затем вычисляем значение выражения скобки с вложенностью на 1 меньше и умножаем ее на x и т.д., пока не «раскрутим» все скобки.

Реализация алгоритма такая:

```
S:=x+2;
for i:=3 to 11 do
  S:=S*x+i;
```

При такой реализации мы затратим лишь 9 операций умножения и 10 сложения (+ операции, связанные с организацией цикла). Этот метод применим не только для многочлена, значение которого мы вычисляем в этом упражнении. Метод можно записать в гораздо более общей форме:

$a_0+a_1(x-b)+a_2(x-b)^2+...+a_n(x-b)^n=a_0+(x-b)(a_1+(x-b)(a_2+...(a_n+(x-b))))$

Это и есть наиболее оптимальный алгоритм для вычисления многочленов в общем случае.

4 вариант: оказывается, можно обойтись вообще без применения оператора цикла. Чтобы задача из пункта б) не обижалась за то, что мы обошли ее вниманием, решим именно ее. Преобразуем выражение следующим образом:

$S=11x^{10}+10x^9+...+2x+1=$

$x^{10}+x^9+...+x^2+x+1+$

$x^{10}+x^9+...+x^2+x+$

$x^{10}+x^9+...+x^2+$

$...+$

x^{10}

Теперь, подсчитав суммы геометрических прогрессий в каждой из строчек (с первым членом

x^{10} и основанием x^{-1}), получим:

$$S=\frac{x^{10}(x^{-11}-1)}{x^{-1}-1}+\frac{x^{10}(x^{-10}-1)}{x^{-1}-1}+...+\frac{x^{10}(x^{-1}-1)}{x^{-1}-1}=\frac{1-x^{11}+(x-x^{11})+...+(x^{10}-x^{11})}{1-x}=\frac{1+x+x^2+...+x^{10}-11x^{11}}{1-x}$$

В числителе снова прогрессия, поэтому: $S=\frac{1}{1-x}\left(\frac{1-x^{11}}{1-x}-11x^{11}\right)=\frac{1-x^{11}(12-11x)}{(1-x)^2}$

С помощью такого преобразования мы получили простую формулу для вычисления S.

Правда, остается вопрос о вычислении x^{11}. Для ее вычисления тоже надо меньше, чем 10 умножений: $x^{11} = x \cdot x^2 \cdot \left(\left(x^2\right)^2\right)^2$. Т.е. достаточно 5 умножений. С использованием такой хитрости затраты на вычисление полученного нами выражения будут даже меньше, чем при использовании схемы Горнера (но, разумеется, лишь для последовательности такого частного вида).

Ясно, что теми же рассуждениями можно найти такую сумму: $S = (n+1)x^n + nx^{n-1} ... + 2x + 1$.

5 способ: читатель, владеющий основами матанализа, может найти S следующим образом:

$$S = (n+1)x^n + nx^{n-1} ... + 2x + 1 = \left(x^{n+1} + x^n ... + x^2 + x\right)' = \left(\frac{x(x^{n+1}-1)}{x-1}\right)' = \frac{(n+1)x^{n+2} - (n+2)x^{n+1} + 1}{(x-1)^2}$$

Думаю, что после этого примера вы убедились, что даже в простых задачах есть над чем подумать.

16. Умножив формулы (1) и (2) главы 4 почленно, получим:

$$НОД(a,b) \cdot НОК(a,b) = q_1^{\min\{r_1,c_1\} \cdot \max\{r_1,c_1\}} q_2^{\min\{r_2,c_2\} \cdot \max\{r_2,c_2\}} ... q_f^{\min\{r_f,c_f\} \cdot \max\{r_f,c_f\}}$$

Отсюда, т.к. $\min\{r_i,c_i\} \cdot \max\{r_i,c_i\} = r_i c_i$, получим:

$$НОД(a,b) \cdot НОК(a,b) = q_1^{r_1} q_2^{r_2} ... q_f^{r_f} \cdot q_1^{c_1} q_2^{c_2} ... q_f^{c_f} = ab$$

18. Формула неверна: проверьте ее, например, для чисел 2, 2, 2.

20. Вся проблема состоит в том, что значения слагаемых мы не знаем до того момента, когда пользователь закончит вводить числа. Было бы неплохо сохранять где-то числа, которые мы вводим. Это действительно можно сделать с помощью массива. Однако с ними мы научимся работать только через одну главу.

Задача кажется неразрешимой, однако решение есть, для этого надо просто немного преобразовать то выражение, сумму которого мы ищем.

$nx_1 + (n-1)x_2 + ... + 2x_{n-1} + x_n =$

$x_1 +$

$x_1 + x_2 +$

..................+

$x_1 + x_2 + ... + x_{n-1} +$

$x_1 + x_2 + ... + x_{n-1} + x_n$

Введем вспомогательную переменную $L_n = x_1 + x_2 + ... + x_{n-1} + x_n$. Исходная задача свелась к вычислению суммы $L_1 + L_2 + ... + L_n$.

А теперь все просто: после того, как мы введем x_n, мы вычислим L_n, и прибавим его к переменной S, значение которой и будет ответом задачи.

21. База индукции уже доказана. Теперь пусть $S_k(n) = 1^k + 2^k + ... + n^k$ - многочлен степени $k+1$, $k \le p$.

$$S_{p+1}(n) = 1^{p+1} + 2^{p+1} + ... + n^{p+1} = 1^p \cdot 1 + 2^p \cdot 2 + ... + n^p \cdot n = \sum_{i=1}^{n} \sum_{j=i}^{n} j^p = n \cdot S_p(n) - \sum_{i=1}^{n-1} \sum_{j=1}^{i} j^p = n \cdot S_p(n) - \sum_{i=1}^{n-1} S_p(n-i) \Rightarrow$$

$\Big[$По ПИ $S_p(n)$ - многочлен $(p+1)-$й степени $\Rightarrow$ $S_p(n-i)$ - тоже многочлен степени не выше $(p+1)\Big] \Rightarrow$

$S_{p+1}(n)$ - многочлен порядка p+1. Причем т.к. $S_{p+1}(0) = 0,$ то коэффициент при 0-й степени n равен 0.

Следовательно, $S_p(n) = a_1 n + a_2 n^2 + ... + a_{p+1} n^{p+1}$.

22. Искать $S_p = 1^p + 2^p + ... + n^p$ будем в виде $S_p = a_1 n + a_2 n^2 + ... + a_{p+1} n^{p+1}$, где a_i, $i = \overline{1, p+1}$ - неопределенные коэффициенты. Давайте найдем их. Будем подставлять в формулу для S_p вместо n числа 1,..,p+1. В результате получим следующую систему алгебраических уравнений

$$\begin{cases} 1a_1 + 1^2 a_2 + ... + 1^{p+1} a_{p+1} = 1^p \\ 2a_1 + 2^2 a_2 + ... + 2^{p+1} a_{p+1} = 1^p + 2^p \\ .... \\ (p+1)a_1 + (p+1)^2 a_2 + ... + (p+1)^{p+1} a_{p+1} = 1^p + 2^p + ... + (p+1)^p \end{cases}$$

Эту систему можно переписать, используя матричное представление, следующим образом:

$Wa = M$, где

$$W = \begin{pmatrix} 1 & 2 & ... & p+1 \\ 1^2 & 2^2 & ... & (p+1)^2 \\ ... & ... & ... & ... \\ 1^{n+1} & 2^{n+1} & ... & (p+1)^{p+1} \end{pmatrix}, \quad a = \begin{pmatrix} a_1 \\ a_2 \\ ... \\ a_n \end{pmatrix}, \quad M = \begin{pmatrix} 1^p \\ 1^p + 2^p \\ ... \\ 1^p + 2^p + ... + (p+1)^p \end{pmatrix}$$

W - матрица Вандермонда. Ее определитель, как известно, отличен от 0, значит, ее матрица обратима. Тогда решение запишется в виде $a = W^{-1}M$. Этот вектор a и задает неопределенные коэффициенты.

Можно вывести формулу для вычисления W^{-1} и, соответственно, $W^{-1}M$ в аналитическом виде. Но это не так важно: результат получен, причем довольно красивый.

23. Допустим, что у нас есть каноническое разложение числа $n!$ (полученное по ОТ Арифметики). Ясно, что нули в конце числа $n!$ могут появиться лишь от произведений 5-к и двоек в каноническом разложении. Т.к. степень двойки в каноническом разложении $n!$ будет больше, чем степень пятерки, то выходит, что кол-во нулей, на которое оканчивается $n!$ равно степени 5-ки в каноническом разложении $n!$. Осталось лишь подсчитать эту степень. Количество чисел в $1 \cdot 2 \cdot 3 \cdot ... \cdot n$, делящееся на 5, равно $\left\lfloor \frac{n}{5} \right\rfloor$. Кроме того, среди чисел, кратных 5, есть числа, кратные 25 (их - $\left\lfloor \frac{n}{25} \right\rfloor$), каждое из которых внесет в КР еще по одной 5-ке, а среди них $\left\lfloor \frac{n}{125} \right\rfloor$ чисел, делящихся на 125 и т.д. Значит количество пятерок в каноническом разложении $n!$ равно $\left\lfloor \frac{n}{5} \right\rfloor + \left\lfloor \frac{n}{25} \right\rfloor + \left\lfloor \frac{n}{125} \right\rfloor + ...$

25. Пусть l_k - длина последовательности из чисел, количество цифр в которых $\le k$. Очевидно, что для десятичной системы счисления выполняется условие $l_k = l_{k-1} + 9 \cdot 10^{k-1} \cdot k$. Значит $l_n = 9\sum_{k=1}^{n} k10^{k-1}$. Если применить прием из задачи 20, то получим, что $l_n = \frac{(9n-1)10^n + 1}{9}$. Отсюда легко найти, что $f(1987) = 1984$.

26. $a_{n+1} = qa_n + d$.

$$S_n = a_1 + (qa_1 + d) + (q^2 a_1 + qd + d) + ... + (q^{n-1}a_1 + q^{n-2}d + ... + qd + d) = a_1(1 + q + q^2 + ... + q^{n-1}) +$$

$$+d + (qd + d) + ... + (q^{n-2}d + ... + qd + d) = \frac{a_1(1-q^n)}{1-q} + d\left(1 + (1+q) + (1+q+q^2) + ... + (1+q+q^2+...+q^{n-2})\right) =$$

$$= \frac{a_1(1-q^n)}{1-q} + d\left((n-1) + (n-2)q + ... + 2q^{n-3} + q^{n-2}\right)$$

Ответ: $S_n = \frac{a_1(1-q^n)}{1-q} + \frac{d(q^n - 1 + n(1-q))}{(1-q)^2}$, $q \neq 1$ (при $q = 1$ получается арифметическая прогрессия).

27. а) От противного: пусть $\sqrt[k]{p^l}$ - представимо в виде несократимой дроби: $\sqrt[k]{p^l} = \frac{m}{n} \Rightarrow m^k = p^l n^k \Rightarrow m^k \vdots p^l$. Т.к. p - простое, то и $m \vdots p$. Подставляя $m = ps$, получим, что и $n \vdots p$. Пришли к противоречию, что доказывает иррациональность числа $\sqrt[k]{p}$.

б) аналогично пункту а).

28. Пусть $\log_a b = x$. Тогда $a^x = b$. Пусть $a = p_1^{\alpha_1} p_2^{\alpha_2} ... p_n^{\alpha_n}$. Ясно, что в каноническом разложении числа b должны участвовать те же числа, что и в разложении числа a. Можно записать:

$$a^x = p_1^{x\alpha_1} p_2^{x\alpha_2} ... p_n^{x\alpha_n} = p_1^{\beta_1} p_2^{\beta_2} ... p_n^{\beta_n} = b \qquad (*)$$

Предположим, что $x = \frac{m}{n}$. Тогда $p_1^{m\alpha_1} p_2^{m\alpha_2} ... p_n^{m\alpha_n} = p_1^{n\beta_1} p_2^{n\beta_2} ... p_n^{n\beta_n}$.

Итак: должно выполняться $m\alpha_i = n\beta_i,\ i = \overline{1,n}$. Значит, x - рациональное $\Leftrightarrow$ $\frac{\beta_i}{\alpha_i} = const = x\ \forall i = \overline{1,n}$.

Замечу, что из (*), вообще говоря, не следует, что $x\alpha_i = \beta_i,\ i = \overline{1,n}$ (подумайте, почему).

Глава 6

6. Доказывать будем по индукции:

БИ: $P_0(x) = 1$ - четная функция, $P_1(x) = x$ - нечетная.

ПИ: пусть для $k = \overline{0, n-1}$ $P_k(x)$ будет четной функцией при четных k, и нечетной функцией при нечетных k.

ШИ: т.к. $P_n(x) = 2xP_{n-1}(x) - P_{n-2}(x)$, то из ПИ следует, что $P_{n-2}(x)$ и $P_{n-1}(x)$ разной четности. Но тогда $2xP_{n-1}(x)$ и $P_{n-2}(x)$ будут одинаковой четности. А отсюда следует, что $P_n(x)$ будет функцией той же четности, что и $P_{n-2}(x)$. Если n - четное, то и $n-2$ четное, и $P_{n-2}(x)$- четная $\Rightarrow P_n(x)$ - четная. Для нечетного n доказательство проводится аналогично.

7. Докажем первое утверждение: если $f(-x) = -f(x)$, то

$$\int_{-l}^{l} f(x)dx = \int_{-l}^{0} f(x)dx + \int_{0}^{l} f(x)dx = [\text{В 1-м инт. } y = -x] = -\int_{l}^{0} f(-y)dy + \int_{0}^{l} f(x)dx = -\int_{0}^{l} f(y)dy + \int_{0}^{l} f(x)dx = 0$$

Чтобы доказать второе утверждение, докажем, что $g(x)T_m(x)T_n(x)$ - нечетная функция при $m+n$ нечетном ($g(x)$ - четная функция). Действительно:

$$g(-x)T_m(-x)T_n(-x) = (-1)^{m+n} g(x)T_m(x)T_n(x) = \left[(m+n) = 2k+1\right] = -g(x)T_m(x)T_n(x).$$

Интересно то, что если $g(x)=\frac{1}{\sqrt{1-x^2}}$, то можно доказать более сильное утверждение: $\int_{-1}^{1} g(x)T_m(x)T_n(x)dx=0$ при любых $m\neq n$ (это свойство называется ортогональностью).

9. Из утверждения задачи 6 следует, что в массиве коэффициентов многочлена Чебышева $P_{n-1}(x)$ будут чередоваться нулевые и ненулевые коэффициенты. Причем мы заранее знаем, где будут находиться нули, а значит, мы можем использовать эти места, чтобы хранить в них коэффициенты многочлена $P_{n-2}(x)$, т.е. одного массива для решения задачи вполне достаточно. Только не забудьте, что при печати ответа надо печатать лишь элементы массива соответствующей четности.

23. Алгоритм (сортировка слиянием) такой: делим массив на 2 части, к каждой из них применяем сортировку слиянием, а затем сливаем 2 массива в один с помощью алгоритма, использованного при решении задачи 17.

В таком виде алгоритм можно написать, только зная, что такое рекурсия. Поэтому переформулируем алгоритм так: сначала разбиваем массив на двойки элементов и упорядочиваем их. Затем сливаем двойки в четверки, далее – четверки в восьмерки и т.д., пока не получим упорядоченный массив.

Подсчитаем количество действий:

На каждом шаге (т.е. слиянии пар подмассивов в больший подмассив) количество действий будет $c_i n$ (c_i могут быть различны в зависимости от шага, но от n – не зависят).

Количество шагов как несложно видеть, равно числу k, такому, что $2^k=n$, т.е. $k=\log_2 n$. Следовательно, общее количество действий = $cn\log_2 n$, где c - некоторая константа, полученная усреднением c_i.

Глава 7

11. Давайте поэкспериментируем для небольших сумм денег: если банкомат может выдать необходимую сумму, то будем писать +, в противном случае -.

1 -; 2 -;3 +;4 -;5 +;6 +;7 -;8 +;9 +;10 +;

А теперь смотрите: сумму в 8, 9 или 10 тугриков банкомат выдать может. Значит, если надо выдать сумму, большую 10, то он может выдавать тройки до тех пор, пока сумму, которую ему осталось выдать, не станет равным 8, 9 или 10 тугрикам, а ее он выдать может. Вот и все решение: оно не требует даже использования циклов.

12. Можно доказать, что наибольшее число из первой тройки идущих подряд сумм, которые может выдать банкомат, равно $2k$. Сначала покажем, что числа $2k-2$, $2k-1$, $2k$ банкомат может выдать.

Ясно, что $k=3s+d$, d - или 1, или 2.

Если $d=1$, то тройка примет вид: $(6s,\ 6s+1,\ 2k)=(3\cdot 2s, 3\cdot s+k, 2k)$.

Если $d=2$, то тройка примет вид: $(6s+2,\ 6s+3,\ 2k)=(3s+k, 3\cdot(s-1)+k, 2k)$.

Значит, банкомат суммы $2k-2, 2k-1, 2k$ выдать может.

Докажем теперь, что нет тройки чисел, которые банкомат мог бы выдать, и которые были бы меньше $2k-2, 2k-1, 2k$.

Докажем от противного. Пусть такая тройка существует. Одно и только одно число из этих трех делится на 3. Остальные тогда должны быть представлены в виде $k+3\cdot a$. Но

тогда они будут различны не менее, чем на 3, что невозможно. Значит тройка $2k-2, 2k-1, 2k$ - минимальна.

13. Доказательство, абсолютно аналогичное приведенному в задаче 12 показывает, что если $НОД(x,y)=1$, то все суммы, начиная с $(x-1)(y-1)$, выдать можно. Если же $НОД(x,y)=c$, то очевидно, что суммы, не кратные c, не могут быть выданы банкоматом, а все суммы, начиная с $c(x-1)(y-1)$, которые кратны c, банкомат может выдать.

14. Сам алгоритм элементарен. Пусть numm – сумма, которую надо выдать, Munz (Münze - монета) – массив достоинств монет, отсортированный по возрастанию (в нашем случае Munz=[1,2,5,10,25,50]), k – количество элементов в Munz.

Процедура по известным k, Munz, numm строит массив M, смысл которого такой: M[i] – количество монет достоинством Munz[i], которые надо выдать.

```
procedure Gierig(var Munz,M:Mas;k,numm:integer);{gierig - жадный}
var
  i:integer;
begin
  for i:=1 to k do
    m[i]:=0;
  for i:=k downto 1 do
    begin
    m[i]:=numm div munz[i];
    numm:=numm mod munz[i];
    end;
end;
```

Если этот алгоритм верен, то очевидно, что он будет и оптимальным. Осталось доказать, что этот алгоритм приведет нас к цели.

Давайте посмотрим, сколько монет достоинством 1 может быть выдано. Если их выдается 2, то ясно, что их можно заменить одной монетой достоинства 2. Но согласно условию задачи, мы должны выдать сумму минимальным количеством монет, значит, количество монет достоинством 1, может быть либо 0, либо 1. Проводя аналогичные рассуждения для других монет, получим:

1	2	5	10	25	50
1	2	1	2	1	Любое

В таблице во второй строке приведены максимальные количества монет достоинства которых находятся в соответствующих позициях 1-й строки.

Приведенный выше алгоритм будет верен не для всех комбинаций монет. Вы можете подумать, какие должны быть ограничения на достоинства монет, чтобы алгоритм дал правильный ответ. Алгоритм был назван gierig, т.к. он – принадлежит классу алгоритмов, называемых жадными.

17. Так как p - простое, то p будет взаимно просто с любым числом, кроме 1. Значит $\varphi(p)=p-1$. Число n будет взаимно просто с p^e, если оно не делится на число p. Значит $\varphi\left(p^e\right)=p^e-p^{e-1}$.

18. $k(rs)=f(rs)g(rs)=f(r)g(r)f(s)g(s)=k(r)k(s)$

19. $\varphi(n)$ - мультипликативна, так как rs взаимно просто с n тогда и только тогда, когда r взаимно просто с n и s взаимно просто с n. А вычислять $\varphi(n)$ можно следующим способом:

$$\varphi(n)=\varphi\left(p_1^{a_1}p_2^{a_2}...p_k^{a_k}\right)=\varphi\left(p_1^{a_1}\right)\varphi\left(p_2^{a_2}\right)...\varphi\left(p_k^{a_k}\right)=\prod_{i=1}^{k}\left(p_i^{a_i}-p_i^{a_i-1}\right)=\left(\prod_{i=1}^{k}p_i^{a_i}\right)\prod_{i=1}^{k}\left(1-\frac{1}{p_i}\right)=n\prod_{i=1}^{k}\left(1-\frac{1}{p_i}\right)$$

Глава 8

1. Пусть a - сторона треугольника, S - площадь треугольника.

$$S_{Koch}=S+3\frac{S}{9}+3\cdot4\frac{S}{9^2}+3\cdot4^2\frac{S}{9^3}+...+3\cdot4^{n-1}\frac{S}{9^n}=\frac{S}{3}\left(4+\sum_{i=1}^{\infty}\left(\frac{4}{9}\right)^i\right)=\frac{S}{3}\left(4+\frac{4/9}{1-(4/9)}\right)=\frac{8S}{5}$$

Подставляя сюда $S=\frac{\sqrt{3}}{2}a^2$, получим $S_{Koch}=\frac{4\sqrt{3}}{5}a^2$

2. Если $n\vdots2$, то последовательно переставляем диски на первый подходящий колышек по направлению часовой стрелки, причем один и тот же диск 2 раза подряд перемещать нельзя. Если же n – нечетное, то переставлять диски надо против часовой стрелки.

3. Пусть $Um(n)$ - количество действий, необходимых для переноса пирамиды из n дисков (нем. Umstellung – «перемещение»). Из алгоритма $Um(n)=2Um(n-1)+1$, $Um(0)=0$.
$Um(n)=2(Um(n-1)+1)-1=2(2(Um(n-2)+1)-1+1)-1=4(Um(n-2)+1)-1=...=2^n-1$.
Поэтому монахам придется сделать $2^{64}-1$ перемещений, прежде, чем наступит конец света. Поэтому конец света наступит нескоро.

Любопытно сравнить легенду о Ханойских башнях с другой восточной легендой. Шахматы были придуманы в Индии, и когда индийский царь узнал об этой игре, он был поражен разнообразием возможных положений фигур. Узнав, что игра была придумана одним из его подданных, царь приказал вызвать изобретателя, и щедро наградить его. Ученый, к удивлению царя, отказался от золота, а вместо этого попросил о следующем: пусть за 1-ю клетку ему выдадут 1 пшеничное зерно, за 2-ю клетку доски – 2 зерна, за 3-ю – 4, и т. д. за n-ю клетку - 2^{n-1} зерен. Царь распорядился выполнить просьбу изобретателя, но когда придворные математики сосчитали количество зерен, которые захотел ученый, то поняли, как провел его мудрец.

А привел я эту легенду (в очень сокращенном варианте) потому, что общее количество зерен, которые должен был выдать царь, было в точности $2^{64}-1$.

4. Можно доказать утверждение и по индукции, но логичнее вывести его непосредственно из определения:

$$C_{n-1}^{m}+C_{n-1}^{m-1}=\frac{(n-1)!}{m!(n-m-1)!}+\frac{(n-1)!}{(n-m)!(m-1)!}=\frac{(n-m)(n-1)!+m(n-1)!}{(n-m)!m!}=\frac{n!}{m!(n-m)!}=C_n^m$$

5. $a^n=a^{d_0+2d_1+2^2d_2+...+2^sd_s}=a^{d_0}\left(a^2\right)^{d_1}\left(a^4\right)^{d_2}...\left(a^{2^s}\right)^{d_s}$. Количество множителей в этом произведении = $[\log_2 n]+1$.

Можно легко построить и рекурсивный алгоритм: $n=2a+b$, где $b=n\bmod 2$. Тогда $x^n=x^b\cdot\left(x^2\right)^a$.

9. Одной из таких подпрограмм является следующая:

```
procedure Smth;
var x:integer;
begin
  x:=random(10);
  if (x<>5) then
    exit;
```

```
  Smth;
end;
```

10. Если программа должна обязательно вызвать себя хотя бы один раз, то процедура, написанная в качестве решения к предыдущей задаче уже не подходит. Но решение все равно существует. Но теперь надо использовать данные, не являющиеся для подпрограммы локальными, например, глобальные переменные.

Глава 10

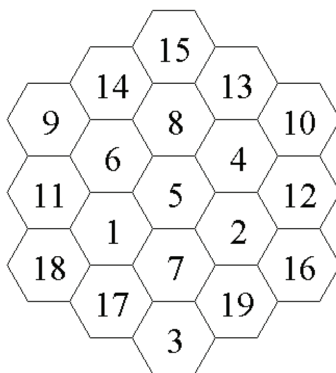
17. Во-первых, заметим, что в квадрате, который получился после замены чисел k на M_k, все элементы различны. Это следует из того, что элементами M_0 являются числа $1,2,...,n^2$, элементами M_1: $n^2+1, n^2+2,...,n^2+n^2$ и т.д., всего таких квадратов n^2, поэтому в результате в полученном суперквадрате будут все числа от 1 до n^4. Докажем, что квадрат – магический.

Магическая сумма в сдвинутом квадрате M_k равна $S_k = S_0 + (k-1)n^3$, где $S_0 = \frac{(n^2+1)n}{2}$ - магическая сумма обычного МК. Теперь давайте подсчитаем магическую сумму в тех строках суперквадрата, которые были получены из строки исходного квадрата, в которой находились числа $k_1, k_2, ..., k_n$:

$$S_* = S_{k_1} + S_{k_2} + ... + S_{k_n} = ... = nS_0 + (S_0 - n)n^3 = \frac{(n^2+1)n^2}{2} + \left(\frac{(n^2+1)n}{2} - n\right)n^3 = \frac{(n^4+1)n^2}{2}$$

Т.е. магическая сумма совпадает для любых строк, столбцов и диагоналей, и равна нужному числу.

18.



Глава 11

Давайте сделаем больше: покажем, как из любую программу изменить так, чтобы она кроме всех своих действий в конце работы печатала себя на экран.

Пусть программа (исходная) выглядит так:

```
строка1
...
строка n
```

Т.к. использовать файл запрещено, то логично, что надо занести в массив все имеющиеся в ней строки.

Т.е. надо ввести массив и переменную, с помощью которой мы пробежимся по нему, (размер массива подсчитаем позже). Итак, добавим в начало программы (после uses) такие строки:

```
const
  nrep=
  SchrLan=10;
var
  Arep:array[1..nrep] of string;
  irep:integer;
```

Теперь в конце программы надо поставить операторы присваивания (очевидно, что 6 строк, которые входят в шапку, тоже надо внести в массив):

```
:
Arep[1]:='1-я строка';
Arep[2]:='2-я строка';
…
Arep[n+6]:='n+6-я строка';
```

Кроме того, затем мы напишем еще SchrLan строк, которые распечатают программу:

```
Arep[n+6+1]:='n+6+1-я строка';
…
Arep[n+6+SchrLan]:='n+6+ SchrLan-я строка';
```

Теперь добавим строки, которые печатают содержимое массива на экран (В SchrLan хранится их количество):

```
for irep:=1 to nrep-SchrLan do
  writeln(Frep,Arep[irep]);
```

Настал черед строк, в которых содержимое программы заносится в массив:

```
for irep:=1 to nrep do
  begin
  write(Frep,'Arep[',irep,']:=');
  writeln(Frep,Arep[irep]);
  end;
```

Но ведь надо еще распечатать и саму часть, которая ответственна за печать:

```
for irep:=nrep-SchrLan+1 to nrep do
  writeln(Frep,Arep[irep]);
end.
```

Глава 18

6. $x(t) = v_0 \cos(\alpha) t$, $y(t) = -\frac{g}{2} t^2 + v_0 \sin(\alpha) t$

7. Надо решить 2 задачи Коши:

$$\begin{cases} \ddot{x}(t) = -kg\dot{x}(t) \\ \dot{x}(0) = v_0 \cos(\alpha) \\ x(0) = 0 \end{cases} \text{ и } \begin{cases} \ddot{y}(t) = -kg\dot{y}(t) - g \\ \dot{y}(0) = v_0 \sin(\alpha) \\ y(0) = 0 \end{cases}$$

Ответ: $x(t) = \frac{v_0 \cos(\alpha)}{kg}(1 - e^{-kgt})$, $y(t) = \frac{1}{kg}\left(v_0 \sin(\alpha) + \frac{1}{k}\right)\left(1 - e^{-kgt}\right) - \frac{t}{k}$

8. Пока тело движется под действием реактивной силы и силы тяжести, то законы движения будут такие:

$\begin{cases} a_x = ku\cos(\alpha) \\ a_y = ku\sin(\alpha) - g \end{cases}$, где a_x, a_y - ускорения тела по осям x, y соответственно.

$t_e = \frac{\ln 2}{k}$ - время, через которое сгорит топливо. После момента t_e тело движется по законам задачи 6: $x(t) = v_e\cos(\alpha)t + x\left(t_e\right)$, $y(t) = -\frac{g}{2}t^2 + v_e\sin(\alpha)t + y(t_e)$, $v_e = v(t_e)$.

9. Размерность множества Кантора равна $\log_2 3$.

10. Размерность пыли Кантора равна 1.

11. Пусть количество итераций будет k, а количество букв F в $neuf$ будет равно n. Улучшать алгоритм можно по-разному. Например, можно вспомнить улучшенный алгоритм вычисления степени, и догадаться, что подставлять в аксиому можно не обязательно всегда по одной neuf, а $neuf^2$ (т.е. строку, получающуюся подстановкой neuf саму в себя), $neuf^4$ и т.д.

Однако самый лучший алгоритм – еще проще. Вместо того чтобы подставлять в аксиому neuf надо просто вычислить $neuf^k$ по следующему алгоритму:

```
erg:=neuf
s:=neuf
for i:=1 to k do
  begin
  s:= подстановка erg в s;
  erg:= s;
  s:=neuf;
  end;
```

Т.е. подставлять надо всегда промежуточный результат в neuf. Количество подстановок будет равно $O(kn)$ (мы считаем, что количество действий, которое необходимо для подстановки строки не зависит от длины этой строки). После того, как вычислено $erg = neuf^k$, надо его подставить в аксиому.

Список литературы

(!!) – книги, которые я рекомендую прочитать в первую очередь.
(!) – полезные книги, которые глубже рассматривают темы, которые были затронуты в этом учебнике.

1. Айра Пол. Объектно-ориентированное программирование на С++, 2-е изд. М.: «Невский диалект» - «Издательство БИНОМ», 1999 г. – 462 с., ил.

Автор определил цель этого ученика так: это не учебник по С++, и не учебник по ООП «вообще», - его цель – научить читателя писать на С++ ОО-программы.

2. Анисимов А. В. Информатика. Творчество. Рекурсия. – Киев: Наукова Думка, 1988.

В книге содержится много примеров рекурсии в литературе.

3. Бентли Джон. Жемчужины программирования. 2-е изд. – СПб.: Питер, 2002. – 272 с.: ил.

В книге на конкретных примерах показывается, как можно оптимизировать код, уменьшать объем требуемой памяти, и.т.д. Примеры действительно очень показательны.

4. Боровский А. Н. Программирование в Delphi 2005. – СПб.: БХВ-Петербург, 2005. – 448 с.:ил.

Книга посвящена разработке различных видов приложений под Delphi. Также в ней описаны архитектура .NET, а также ADO.NET, ASP.NET и другие технологии программирования.

5. Брукшир Дж. Гленн. Введение в компьютерные науки. 6-е изд. – М.: Издательский дом «Вильямс», 2001. – 688 с.: ил.

Книга является введением во все основные разделы компьютерных наук. Но я думаю, что человек, который еще не писал ни одной программы, не прочувствует многих идей, которые описаны в книге, а тому, кто уже опытен в программировании, эта книга нужна в меньшей степени.

6. Гарднер Мартин. Математические досуги: Москва «Мир», 2000.

Мартин Гарднер – известный популяризатор математики. В этой книге собрано много интересных задач. В ней я увидел задачу о правильном шестиугольнике (см. главу Матрицы).

7. Гради Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на С++

Книга по ООП «вообще». Комментарий тот же, что и к книге [5]: программист, хорошо владеющий ООП, будет рассматривать большую часть книги как сборник прописных истин, а тот, кто не писал ни одной ОО-программы, не научится это делать. Но в приложениях есть хорошая диаграмма, на которой показано развитие ОО-языков, а также приведены характеристики наиболее популярных языков.

8. (**!!**) Докинз Ричард Эгоистичный ген: Пер. с англ. – М.: Мир, 1993.- Д63 318 с., ил.

Потрясающая книга, в которой автор развивает концепцию, согласно которой естественный отбор действует не на уровне видов, популяций, или отдельных организмов, а на уровне генов. Беспощадность Докинза можно оценить по следующей цитате, взятой из предисловия: «Мы всего лишь машины для выживания, самоходные транспортные средства, слепо запрограммированные на сохранение эгоистичных молекул, известных под названием генов».

9. (**!**) Кнут Дональд Эрвин Искусство программирования (в 3 т.) М. Вильямс,2002

Всемирно известная книга по вычислительной математике. В ней подпробно рассматриваются важнейшие алгоритмы и структуры данных. Но чтение книги затруднено тем, что автор использует язык ассемблера, придуманный специально для этой книги.

10. (**!!**) Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: Построение и анализ – М. МЦНМО, 2002. – 960 с.: 263 ил.

Один из лучших учебников по теории алгоритмов. В книге рассматриваются важнейшие алгоритмы и анализируются характеристики алгоритмов. Изложение сопровождается примерами программ с использованием псевдокода (унифицированного языка наподобие Паскаля). Очень полезная книга.

11. (**!**) Кроновер Ричард М.: Фракталы и хаос в динамических системах. Основы теории. Москва: Постмаркет, 2000. – 352 с.

Введение во фрактальную геометрию. Учебник рассчитан на студентов, но многие главы может читать и старшеклассник. Большое внимание уделено приложениям теории фракталов.

12. (**!**) Б. Майер. Объектно-ориентированное программирование. Концепции разработки. М.: Русская редакция, 2004.

Рассматриваются многие важные вопросы ООП и программирования параллельных процессов, а также вкратце рассматриваются важнейшие ОО-языки.

13. (**!**) Новиков Ф.А. Дискретная математика для программистов. – СПб.: Питер, 2002. – 304 с.:ил.

Название книги отражает ее содержание. Кроме основ дискретной математики приводятся применения дискретной математики для построения многих важных алгоритмов.

14. (**!!**) Пенроуз Роджер. Новый ум короля: О компьютерах, мышлении и законах физики – М.: Едиториал УРСС, 2003. – 384 с.

Очень интересная книга выдающегося математика и физика. Пенроуз считает, что умственная деятельность человека не сводится к алгоритмическим вычислениям и на протяжении всей книги обосновывает свою точку зрения с позиций математики, физики, биологии и теории алгоритмов.

15. (!) Пинкер Стивен. Язык как инстинкт: пер. с англ./ Общ. ред. В.Д.Мазо. – Едиториал УРСС, 2004, - 456 с.

Интересная книга, посвященная исследованиям человеческого языка. Многие поразительные факты автор объясняет, рассматривая язык как не более чем инстинкт. Пинкер пишет: «Люди знают, как говорить, примерно в том же смысле, как пауки знают, как плести паутину».

16. Пойа Джордж. Математическое открытие. Решение задач: основные понятия, обучение и преподавание. М.: Наука, 1976 г., 448 стр. с илл.

Интересная книга известного математика и педагога. В ней автор анализирует, как математик анализирует задачу, решает ее и обобщает результаты.

17. (!!) Таненбаум Эндрю. Современные операционные системы. 2-е изд. – СПб.: Питер, 2002. – 1040 с.:ил.

В книге рассматривается структура операционных систем. Подробно рассматриваются проблемы взаимоблокировки процессов, строения файловых систем. Большое внимание уделено рассмотрению мультипроцессорных операционных систем, безопасности. Автор подробно разбирает реализацию общих принципов построения ОС на примере Windows 2000 и Unix.

18. Фаронов В.В. Delphi. Программирование на языке высокого уровня: Учебник для вузов: СПб.: Питер, 2003. – 640 с.:ил.

Может быть полезен как справочное руководство по Delphi.

19. (!!) Хофштадтер Д. ГЁДЕЛЬ, ЭШЕР, БАХ: эта бесконечная гирлянда. – Самара: Издательский Дом «Бахрах-М», 2001. – 752 с.

Тематика книги Хофштадтера во многом перекликается с книгой [14]. Автор анализирует мышление человека и его познавательную деятельность с позиций математики, программирования, биологии, психологии, дзен-буддизма и физики. Интересно, что Хофштадтер – сторонник того, что любое устройство, способное к алгоритмическим действиям, может обладать интеллектом.

20. (!!) Хофштадтер Д., Деннетт Д. ГЛАЗ РАЗУМА. Самара: Издательский Дом «Бахрах-М», 2003. – 432 с.

В книге собраны произведения многих авторов о мышлении и интеллекте. Авторы после каждого произведения рассуждают на темы, затронутые в произведении. Книга зачастую поражает воображение, когда простые на первый взгляд вопросы становятся глубокими философскими проблемами.

21. (!) Шрёдер Манфред. Фракталы, хаос, степенные законы. Миниатюры из бесконечного рая. – Ижевск: НИЦ «Регулярная и хаотическая динамика», 2001, 528 стр.

Книга, позволяющая читателю глубже осознать смысл самоподобия. В книге рассматриваются применения самоподобия в различных науках.

22. (!) Западноевропейский эпос. Составитель – Плотникова Л.А. Ленинград. Лениздат 1977.

В этой книге содержатся самые известные эпические поэмы средневековья: «Беовульф», «Старшая Эдда», «Песнь о Нибелунгах», «Песнь о Роланде», «Песнь о Сиде».

23. **(!)** Falconer Fractal Geometry. Mathematical Foundations and Applications: John Wiley&Sons, 1990

Монография по фрактальной геометрии и ее приложениям.

Мирончеко Андрей Станиславович

Императивное и объектно-ориентированное
программирование на Turbo Pascal и Delphi

Главный редактор Мирончеко А.С.
Литературный редактор Мирончеко А.С.
Дизайн Мирончеко С.С., Мирончеко С. Г.
Верстка Мирончеко А.С.

ISBN 978-966-413-039-1

Подписано в печать 20.07.2007
Формат 70*100/16. Бумага офсетная.
Печать цифровая.
Заказ № 03128. Тираж 200 экз.
Отпечатано с готового оригинал-макета в типографии «ВМВ».
Украина, 65053, Одесса, пр-т Добровольского, 82-а. Тел. 751-14-87.