Д. М. Ушаков, Т. А. Юркова

ПАСКАЛЬ
для школьников

2-е издание

ПИТЕР®

Д. М. Ушаков, Т. А. Юркова

ПАСКАЛЬ ДЛЯ ШКОЛЬНИКОВ

2-е издание

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2013

ББК 32.973.2-018.1я7
УДК 004.43(075)
У93

Д. Ушаков, Т. Юркова

У93 Паскаль для школьников. 2-е изд.. — СПб.: Питер, 2013. — 320 с.: ил.

ISBN 978-5-496-00440-4

Эта книга — не учебник, а скорее помощник в освоении языка программирования Паскаль, с которым на уроках информатики знакомятся все школьники. Она состоит из уроков, посвященных практическим вопросам программирования и решения задач. Многочисленные примеры позволяют лучше понять, как разработать алгоритм, написать собственную программу, правильно оформить ее текст. Советы и примечания помогают читателю обратить внимание на важные детали, позволяя избежать подводных камней и более эффективно писать программы.

Книга подготовлена преподавателями информатики в школе, имеющими большой опыт многолетней практической работы.

Во второе издание добавлено несколько новых глав, посвященных записям, динамическим переменным, стеку, очереди и спискам. Также освещена одна из самых сложных тем в программировании — построение рекурсивных алгоритмов.

6+ (В соответствии с Федеральным законом от 29 декабря 2010 г № 436-ФЗ.)

ББК 32.973.2-018.1я7
УДК 004.43(075)

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-5-496-00440-4

© ООО Издательство «Питер», 2013

Содержание

Предисловие ко второму изданию 15

Вступление 16

От издательства 16

ТЕМА 1. Как написать простую программу на Паскале 17

Урок 1.1. Выводим сообщение на экран дисплея 18

Урок 1.2. Как заложить эту программу в компьютер 19

Этапы создания компьютерной программы 20

Урок 1.3. Оформление текста на экране 28

Выводы 34

Контрольные вопросы 34

ТЕМА 2. Как включить в работу числовые данные 36

Урок 2.1. Начнем с простого: целые числа 37

Понятие переменной 38

Тип Integer. Оператор присваивания. Вывод на экран 38

Операции с типом Integer 40

Стандартные функции типа Integer 42

Как представляются переменные целого типа в памяти компьютера 43

Урок 2.2. Включаем в работу вещественные числа 45

Описание вещественного типа данных (real) 45

Форматы записи вещественных переменных 46

Вещественные операции 46
Стандартные функции типа real 47
Запись математических выражений 48
Как представляются переменные вещественного типа в памяти компьютера 50
Урок 2.3. Как совместить переменные целого и вещественного типа 51
Преобразование типов 51
Правила приоритета в выполняемых действиях 52
Действия над данными разных типов 53
Урок 2.4. Ввод и вывод данных 56
Вводим переменные с клавиатуры 57
Красивый вывод на экран 57
Задание значений переменных датчиком случайных чисел 61
Урок 2.5. Зачем нужны константы в программе? 62
Выводы 64
Контрольные вопросы 64

ТЕМА 3. Учимся работать с символами 66

Урок 3.1. Как компьютер понимает символы 67
Кодовая таблица ASCII 67
Описание типа Char и стандартные функции 68
Урок 3.2. Тип Char — порядковый тип! 70
Выводы 71
Контрольные вопросы 72

ТЕМА 4. Джордж Буль и его логика 73

Урок 4.1. Необходим еще один тип — логический! 74
Логический тип данных (Boolean) 75
Операции отношения 75
Ввод-вывод булевских переменных 76
Урок 4.2. Логические (булевские) операции 76
Логическое умножение (конъюнкция) 76
Логическое сложение (дизъюнкция) 77
Исключающее ИЛИ (сложение по модулю 2) 77
Логическое отрицание (инверсия) 78

Применение логических операций в программе 78
Приоритет логических операций 80
Выводы 81
Контрольные вопросы 81

ТЕМА 5. Анализ ситуации и последовательность выполнения команд 82

Урок 5.1. Проверка условия и ветвление в алгоритме 83
Полная и неполная форма оператора if 84
Оформление программ 86
Урок 5.2. Блоки операторов 88
Урок 5.3. Ветвление по ряду условий (оператор case) 92
Выводы 96
Контрольные вопросы 96

ТЕМА 6. Многократно повторяющиеся действия 98

Урок 6.1. Оператор цикла for 99
Оператор for с последовательным увеличением счетчика 100
Оператор for с последовательным уменьшением счетчика 101
Урок 6.2. Применение циклов со счетчиком 101
Цикл в цикле 102
Трассировка 103
Вычисление суммы ряда 105
Выводы 108
Контрольные вопросы 109

ТЕМА 7. Циклы с условием 110

Урок 7.1. Цикл с предусловием 111
Описание цикла с предусловием 111
Приближенное вычисление суммы бесконечного ряда . . . 112
Возведение числа в указанную целую степень 115
Урок 7.2. Цикл с постусловием 119
Описание цикла с постусловием 120
Использование циклов repeat и while 120
Относительность выбора операторов while и repeat 123
Выводы 129
Контрольные вопросы 129

ТЕМА 8. Массивы — структурированный тип данных 131

Урок 8.1. Хранение однотипных данных в виде таблицы 132
Основные действия по работе с массивами 133
Описание массива на языке Паскаль 133
Заполнение массива случайными числами и вывод массива на экран 134
Создание пользовательского типа данных 137
Поиск максимального элемента массива 140
Вычисление суммы и количества элементов массива с заданными свойствами 144
Урок 8.2. Поиск в массиве 148
Определение наличия в массиве отрицательного элемента с использованием флажка 148
Определение наличия в массиве отрицательных элементов путем вычисления их количества 149
Нахождение номера отрицательного элемента массива . . . 150
Урок 8.3. Двумерные массивы 154
Выводы 156
Контрольные вопросы 157

ТЕМА 9. Вспомогательные алгоритмы. Процедуры и функции. Структурное программирование 158

Урок 9.1. Конструирование алгоритма «сверху вниз» 159
Практическая задача с использованием вспомогательных алгоритмов 160
Урок 9.2. Пример работы с функцией: Поиск максимального элемента 167
Выводы 168
Контрольные вопросы 169

ТЕМА 10. Как работать с символьными строками 170

Урок 10.1. Работаем с цепочками символов: тип String 171
Описание строковой переменной 171
Основные действия со строками 172
Урок 10.2. Некоторые функции и процедуры Паскаля для работы со строками 173

Использование библиотечных подпрограмм работы со строками . . . 173
Выводы . . . 175
Контрольные вопросы . . . 175

ТЕМА 11. Процедуры и функции с параметрами 176

Урок 11.1. Простые примеры использования подпрограмм с параметрами . . . 177
Простейшие процедуры с параметрами . . . 177
Формальные и фактические параметры . . . 179
Простейшие функции с параметрами . . . 179
Урок 11.2. Способы передачи параметров . . . 181
Выводы . . . 183
Контрольные вопросы . . . 184

ТЕМА 12. Файлы: сохраняем результаты работы до следующего раза . . . 185

Урок 12.1. Как работать с текстовым файлом . . . 186
Открытие файла для чтения . . . 186
Открытие файла для записи . . . 188
Урок 12.2. Сохранение двумерного массива чисел в текстовом файле . . . 192
Сохранение числовых данных в текстовом файле . . . 192
Сохранение массива чисел в текстовом файле . . . 192
Дописывание информации в конец файла . . . 196
Выводы . . . 197
Контрольные вопросы . . . 197

Тема 13. Графический режим работы. Модуль Graph 199

Урок 13.1. Включаем графический режим работы . . . 200
Особенности работы с графикой . . . 200
Переключение в графический режим видеоадаптера . . . 201
Урок 13.2. Продолжаем изучать возможности модуля Graph . . . 203
Рисование линий средствами модуля Graph . . . 203
Рисование окружностей средствами модуля Graph . . . 205
Выводы . . . 206
Контрольные вопросы . . . 207

Тема 14. Операторы, изменяющие естественный ход программы 208
Урок 14.1. Использование оператора безусловного перехода goto 210
Урок 14.2. Операторы, изменяющие ход выполнения цикла 213
Оператор break 213
Оператор continue 214
Выводы 215
Контрольные вопросы 215

Тема 15. Группируем данные: записи 216
Урок 15.1. Описание типа данных record 218
Урок 15.2. Когда и как разумно использовать записи 220
Создание собственного типа данных — запись 220
Массив записей 220
Оператор присоединения with 221
Пример выбора структуры данных 223
Записи записей 224
Выводы 225
Контрольные вопросы и задания 225

Тема 16. Динамические переменные 226
Урок 16.1. Выделение памяти 227
Урок 16.2. Адреса 229
Урок 16.3. Указатели 230
Указатели на отдельные переменные 230
Указатели на блоки переменных 232
Урок 16.4. Динамическое выделение памяти 232
New и Dispose 233
Динамическое выделение памяти для массивов 235
GetMem и FreeMem 236
Обращение к элементам массива, созданного динамически 237
Массив переменной длины 238
Выводы 241
Контрольные вопросы 242

Тема 17. Динамические структуры данных. Стек 244
Урок 17.1. Опишем тип данных 245
Урок 17.2. Создание стека и основные операции со стеком 247
Добавление элемента в стек (Push) 248
Извлечение элемента из стека (Pop) 251
Проверка стека на пустоту (StackIsEmpty) 252
Урок 17.3. Использование стека 253
Программирование стека при помощи массива 255
Выводы 256
Контрольные вопросы и задания 256

Тема 18. Динамические структуры данных.
Очередь 258
Урок 18.1. Принцип работы и описание типа данных 259
Урок 18.2. Основные операции с очередью 261
Добавление элемента в очередь (EnQueue). 261
Извлечение элемента из очереди (DeQueue) 263
Проверка очереди на пустоту (QueueIsEmpty) 264
Урок 18.3. Использование очереди 264
Программирование очереди при помощи массива 267
Выводы 269
Контрольные вопросы 269

Тема 19. Динамические структуры данных.
Однонаправленный список 270
Урок 19.1. Описание типа данных и принцип работы 271
Урок 19.2. Основные операции с однонаправленным списком . . . 272
Последовательный просмотр всех элементов списка 272
Помещение элемента в список 273
Удаление элемента из списка 275
Урок 19.3. Обработка списков 276
Целесообразность использования однонаправленного списка 278
Выводы 280
Контрольные вопросы 280

Тема 20. Рекурсия 281
Урок 20.1. Описание принципа 282
Урок 20.2. Ханойские башни 285
Урок 20.3. Структура рекуррентной подпрограммы 287
Урок 20.4. Пример рекуррентного решения нерекуррентной задачи 288
Урок 20.5. Пример рекуррентного решения рекуррентной задачи 289
Выводы 291
Контрольные вопросы 291

Приложение 1. Элементы блок-схем 292

Приложение 2. Задачи 295
Integer. Описание. Ввод. Вывод. Операции 296
Real. Описание. Ввод. Вывод. Операции и функции 296
Real. Запись и вычисление выражений 297
Char. Описание. Ввод. Вывод. Функции 298
Boolean. Запись выражений 298
Boolean. Вычисление выражений 299
If. Простые сравнения. Min/max/средний 300
If. Уравнения и неравенства с параметрами 300
For. Перечисления 300
For. Вычисления со счетчиком цикла 301
For. Перебор со сравнениями 302
While-Repeat. Поиск 302
While-Repeat. Ряды 303
Графика. Прямые 303
Графика. Окружности 304
Массивы. Заполнение, вывод, сумма/количество 305
Массивы. Перестановки 305
Массивы. Поиск 306
Массивы. Проверки 307
Массивы. Максимумы 307
Подпрограммы без параметров 307
Строки. Часть I 308
Строки. Часть II 309

Подпрограммы с параметрами. Часть I 309
Подпрограммы с параметрами. Часть II 310
Подпрограммы с параметрами. Часть III 310
Файлы 311
Однонаправленный список 312
Рекурсия 313

Предисловие ко второму изданию

После выхода первого издания книги к нам стали все чаще обращаться наши коллеги и ученики с просьбой дополнить первое издание информацией о наиболее изучаемых и востребованных структурах данных. В это издание мы добавили несколько глав, посвященных записям, динамическим переменным, стеку, очереди и спискам. Также мы постарались осветить одну из самых сложных тем в программировании — построение рекурсивных алгоритмов.

В приложении мы решили отказаться от сборника домашних заданий с множеством вариантов по нескольким темам. Вместо этого мы поместили в приложение большое число тематических заданий, организованных блоками по 5–8 задач. Задания в каждом блоке расположены от простого к сложному. Мы используем их на наших уроках для организации практических занятий при закреплении теоретического материала (одно занятие — один блок).

Авторы выражают глубочайшую признательность одному из лучших своих учеников, доценту кафедры безопасности информационных систем СПбГУАП, к. т. н. Евгению Михайловичу Линскому за поддержку, множество полезных советов и большую помощь при работе над вторым изданием книги.

Вступление

Что такое язык программирования? Любая задача, которую решает компьютер, записывается в виде последовательности команд. Такая последовательность называется программой. Команды, конечно, должны быть представлены на языке, понятном компьютеру. Один из таких языков — язык программирования Паскаль. Он разработан швейцарским профессором Николаусом Виртом специально для обучения студентов программированию. К особенностям языка относится также и его структурность. То есть программа легко разбивается на более простые, непересекающиеся блоки, те, в свою очередь, на еще более простые блоки. Это также облегчает программирование. В 1979 году язык был утвержден в качестве стандартного. Вирт назвал его в честь французского ученого Блеза Паскаля, изобретателя счетной машины. Язык Паскаль прост, логичен и эффективен. Он получил распространение во всем мире. Материал книги построен на конкретных примерах программ. Длительных теоретических пояснений нет, поэтому крайне необходимо внимательно читать комментарии в текстах программ!

Итак, начинаем обучение сразу с написания первой программы на Паскале.

Желаем удачи!

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все файлы, описываемые в книге, вы сможете найти по адресу http:/www.piter.com/download.

Подробную информацию о наших книгах вы найдете на веб-сайте издательства http:/www.piter.com

ТЕМА 1

Как написать простую программу на Паскале

Изучение Паскаля мы начнем сразу с конкретных примеров, которые помогут вам почувствовать процесс программирования вживую. Мы проведем вас шаг за шагом через все стадии программирования. Мы хотим, чтобы вы сразу ощутили уверенность, чтобы вы поняли: «Если я смог написать одну программу, значит, смогу и вторую!»

Урок 1.1. Выводим сообщение на экран дисплея

Нашей первой программой будет программа-приветствие. Она просто выведет текст на экран компьютера и завершит свою работу. В этом уроке мы рассмотрим основные правила оформления программы.

СОВЕТ

Внимательно читайте комментарии, они указываются в фигурных скобках { }.

Пример 1.1. Первая программа

```
program First; { Первая строка — заголовок программы.
                 program — служебное слово;
                 First — имя нашей программы,
                 его вы придумываете сами.
                 В конце строки стоит
                 точка с запятой.
                 При перечислении инструкций Паскаля
                 между ними нужно
                 ставить ";".
                 Дальше идет тело программы.
                 Оно всегда начинается со слова begin }
begin          { Здесь в конце строки нет
```

```
                   точки с запятой.
                   Следующая команда, или оператор,
                   выводит слово ПРИВЕТ на экран;
                   после вывода курсор остается на той же
                   строке в конце текста;
                   текст для вывода всегда заключается
                   в апострофы.}
   write ('Привет, '); { В конце строки обязательна
                   точка с запятой}
                 { Следующий оператор выведет на экран
                   слово ДРУЗЬЯ! и переведет курсор
                   на следующую строку, так как символы
                   "ln" в операторе writeln означают
                   "line" — по-английски "строка" }
   writeln ('друзья!');
   writeln ('Это вторая строка') { Здесь в конце строки
                   не обязательна точка с запятой, так как
                   это последний оператор.
                   Проще говоря, перед end точку с запятой
                   можно не ставить }
end.             { Словом end кончается
                   тело программы;
                   в конце обязательно стоит точка }
```

ЗАМЕЧАНИЕ

Очень важно понимать, когда нужно ставить точку с запятой, а когда нет. Основное правило — точка с запятой должна ставиться при перечислении инструкций. Так, в рассмотренном только что примере точка с запятой стоит между разделом program *и разделом тела программы, а также в теле программы при перечислении операторов. На последнем операторе* writeln *перечисление заканчивается, поэтому мы не поставили точку с запятой.*

Урок 1.2. Как заложить эту программу в компьютер

В этом уроке мы рассмотрим и проделаем все действия, необходимые для работы в среде Turbo Pascal. Мы условно разделили их на 7 шагов. Может быть, такой длинный процесс покажется вам утомительным. Ну, что ж поделаешь — тяжело в учении — легко в бою.

Обнадежьте себя тем, что эти 7 шагов приведут вас к первому конкретному результату. Это начало большого пути!

Сначала рассмотрим, какие этапы должен пройти пользователь (программист) для того, чтобы увидеть на экране правильные результаты работы своей программы.

Этапы создания компьютерной программы

- ✦ **1 этап.** Редактирование текста программы (команда `Edit`). На этом этапе пользователь набирает свою программу в символах выбранного им языка программирования.
- ✦ **2 этап.** Компиляция программы (команда `Compile`). В результате программа пользователя переводится из символов языка программирования в двоичный код компьютера. При обнаружении ошибок происходит возврат к 1 этапу.
- ✦ **3 этап.** Построение программы (команда `Build`). Подгружаются библиотечные модули[1], и готовая программа в двоичном коде сохраняется на диске. При обнаружении ошибок происходит возврат к 1 этапу.
- ✦ **4 этап.** Запуск программы на выполнение (команда `Run`). При обнаружении ошибок происходит возврат к 1 этапу.

Прохождение всех этих этапов заложено в интегрированной среде Turbo Pascal.

ЗАМЕЧАНИЕ

Почти любая компьютерная программа состоит из блоков (модулей), каждый из которых выполняет какое-то одно действие. Программу составляют из этих блоков, как из кирпичиков. В 9-й теме мы научимся создавать такие блоки самостоятельно. Однако даже в самых простых случаях приходится выполнять действия, без которых не обходится почти ни одна программа — например выводить информацию на экран или вводить что-то с клавиатуры. Так как эти действия нужны всем, для них ввели специальные названия (например, `write` *и* `writeln`*) и заложили непосредственно внутрь программы Turbo Pascal. Они-то и называются библиотечными модулями. Их машинные коды хранятся в специальных файлах (библиотеках) и подсоединяются к вашим программам на этапе построения.*

---

[1] О том, что такое библиотечные модули, см. замечание далее по тексту.

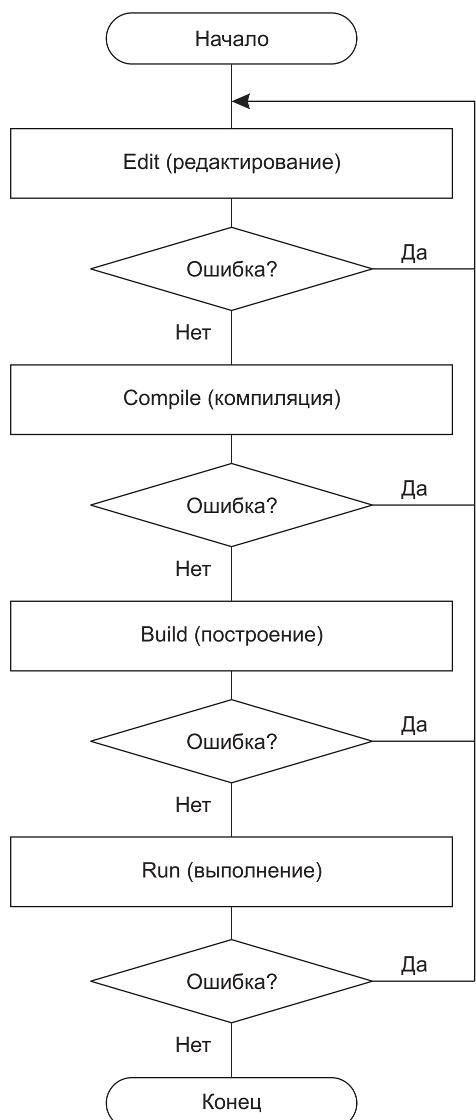


Рис. 1.1. Схема прохождения этапов создания программы на компьютере

Вам предлагается выполнить последовательно следующие действия.

1. Запуск среды Turbo Pascal

- Щелкните на значке Turbo Pascal, если он существует на вашем Рабочем столе, или найдите на диске файл turbo.exe или bp.exe (исполняемые файлы с программой среды Turbo Pascal) с помощью функции поиска Windows (Пуск ▸ Найти).

При успешном запуске интегрированной среды Turbo Pascal вы увидите на экране следующие элементы:

- главное меню (строка сверху);
- окно редактирования (синее поле посередине);
- описания функциональных клавиш (строка внизу).

ЗАПОМНИТЕ!

Клавиша F10 *вызывает главное меню (можно также щелкнуть мышью).*

Клавиша Esc *осуществляет возврат из любого вызванного режима к предыдущему.*

СОВЕТ

Эти советы касаются пользователей Turbo Pascal для Windows.

Скорее всего, вы запускаете Turbo Pascal из Windows, создав для этого ярлык (например, на Рабочем столе). Однако Turbo Pascal — программа для MS-DOS, поэтому в ходе работы с ней вы столкнетесь с несколькими особенностями.

Во-первых, программы для MS-DOS обычно запускаются в полноэкранном текстовом режиме. Это кажется нам неудобным хотя бы потому, что частота мерцания монитора в этом режиме составляет всего 60 Гц, в то время как для безопасности глаз считается необходимым хотя бы 85 Гц. Обычно графический режим Windows настроен на максимальную частоту обновления. Мы рекомендуем вам перейти после запуска программы Turbo Pascal в оконный режим работы (комбинация клавиш Alt+Enter*) и развернуть окно на весь экран.*

Во-вторых, переключатель клавиатуры для MS-DOS-программ не имеет никакого отношения к обычному переключателю клавиатуры Windows.

Запомните: в Turbo Pascal переключение на русскую раскладку клавиатуры производится нажатием правого Ctrl+Shift, *а обратно на латинскую — нажатием левого* Ctrl+Shift. *При этом индикатор клавиатуры на панели задач не меняется — не обращайте на него внимания. Обычно при переключении раскладки для MS-DOS компьютер также издает звуковой сигнал.*

В-третьих, окно программы для MS-DOS крайне не рекомендуется закрывать щелчком на кнопке закрытия окна Windows. Это действие приведет к аварийному принудительному закрытию программы Turbo Pascal и, скорее всего, к потере несохраненной программы. Закрывать Turbo Pascal нужно средствами Turbo Pascal — с помощью команды Exit *в меню* File *или комбинации клавиш* Alt+X *на клавиатуре.*

2. Работа в окне редактирования Edit

❖ Перейдите в главное меню (клавиша F10).

❖ Выберите пункт File с помощью клавиш перемещения курсора (←, →).

❖ Нажмите клавишу Enter.

Появится раскрывающееся меню File:

```
New
Open                    F3
Save                    F2
Save as
Save all
-------------------- ------
Change dir
Print
Printer setup
DOS shell
Exit                    Alt+X
```

Комбинации клавиш, указанные справа от названия команды, дают возможность сразу войти в этот режим.

❖ Выберите команду New с помощью клавиш перемещения курсора (↑, ↓)

❖ Нажмите клавишу Enter.

На экране откроется пустое окно, озаглавленное NONAME00.PAS. Это имя, данное средой по умолчанию вашей будущей программе (точнее, файлу, в котором будет храниться ваша будущая программа). Если вы повторите описанную выше операцию, то раскроется

еще одно окно, но уже с именем NONAME01.PAS. Так можно раскрыть достаточное число окон редактирования. Для переключения окон достаточно, удерживая нажатой клавишу Alt, нажать клавишу с цифрой, представляющей порядковый номер окна. Эту операцию также выполняет клавиша F6.

❖ Итак, курсор находится в левом верхнем углу окна редактирования. Наберите вашу первую программу, но без комментариев. В конце каждой строки нажимайте Enter. Программа будет выглядеть следующим образом:

Пример 1.2. Первая программа без комментариев

```
program First;
begin
   write('Привет, ');
   writeln('друзья!');
   writeln('Это вторая строка')
end.
```

СОВЕТ

Компилятору языка Turbo Pascal безразлично, какие буквы вы используете при наборе программы: строчные или заглавные. В наших примерах вам предлагается стандартный вариант использования регистра символов в программе.

При написании текста программы вы можете свободно вставлять пробелы и перевод строки между независимыми операторами. Но при этом позаботьтесь о том, чтобы вашу программу было легко читать!

При работе в редакторе полезно знать основные комбинации клавиш (табл. 1.1).

Таблица 1.1. Комбинации клавиш редактора Turbo Pascal

Действие	Клавиша
Образование новой строки	Enter
Переход в начало строки	Home
Переход в конец строки	End

Действие	**Клавиша**
Переход в начало файла	Ctrl+Home
Переход в конец файла	Ctrl+End
Включение режима замены символов	Insert (Ins)
Включение режима вставки символов	Insert (Ins)
Удаление текущего символа	Delete (Del)
Удаление предыдущего символа	Backspace (Bs) — правая клавиша в ряду цифровых клавиш
Перемещение курсора по тексту	←, →, ↑, ↓
Страница вверх	Page Up (PgUp)
Страница вниз	Page Dn (PgDn)
Склеивание двух строк	1. Переход в конец 1-й строки 2. Клавиша Delete
Удаление текущей строки	Ctrl+Y

ЗАМЕЧАНИЯ

*Клавишами управления (*PgUp, PgDn, Ins, Del *и т. д.) на вспомогательной цифровой клавиатуре можно пользоваться при выключенном режиме* Num Lock*.*

В главном меню есть пункт Edit*. Раскрывающееся меню, которое появляется при выборе этого пункта, позволяет выполнять различные операции с текстом.*

Выделить фрагмент текста можно с помощью клавиш управления курсором, держа нажатой клавишу Shift*, или путем перетаскивания мыши по выделяемому фрагменту.*

С выделенным фрагментом можно работать почти так же, как в Windows — удалять, вырезать, копировать, вставлять.

В Turbo Pascal, как и в Windows, есть буфер обмена, куда можно скопировать выделенный фрагмент, а потом вставить из него в другое место. Причем вставлять из буфера можно несколько раз. Сочетания клавиш для работы с буфером обмена используются не совсем привычные для Windows (табл. 1.2).

Таблица 1.2. Комбинации клавиш для работы с буфером обмена среды Turbo Pascal

Действие	**Комбинация клавиш**
Удалить выделенный фрагмент	Ctrl+Delete
Вырезать в буфер (Cut)	Shift+Delete
Копировать в буфер (Copy)	Ctrl+Insert
Вставить из буфера (Paste)	Shift+Insert

Задание 1.1. Напишите (в редакторе Turbo Pascal) программу, которая выводит на экран фразу «`Всем привет!`» 80 раз — в таблице из 20 строк по 4 столбца.

Подсказка. Для задания расстояния между колонками используйте несколько пробелов. Напишите сначала только один оператор `write`, который выведет одну фразу (не забудьте про пробелы). Затем скопируйте его еще 3 раза, чтобы получить целую строку. В конце не забудьте поставить переход на следующую строку (`writeln`). После этого скопируйте получившийся текст еще 19 раз. Копировать begin и end не нужно!

3. Сохранение программы в файле на диске

- В главном меню выберите File (клавиши Alt+F).
- В раскрывающемся меню File выберите команду Save as…. (Turbo Pascal откроет окно Save File As.)
- Переключайтесь между элементами окна клавишей Tab (или щелчком мыши).
- В поле ввода Save file as введите имя, под которым собираетесь сохранить файл, например MY.PAS, и нажмите Enter (расширение .pas набирать не обязательно).

ЗАМЕЧАНИЯ

В поле Files *отображаются имена файлов текущего каталога в соответствии с маской, установленной в поле* Save file as*. Когда вы выберете нужный файл в поле* Files *и нажмете* Enter*, то его имя автоматически появится в поле ввода* Save file as*. Кнопка* OK *служит для подтверждения выбранных действий. Кнопка* Cancel *отменяет все действия и закрывает диалоговое окно. Кнопка* Help *выводит окно с подсказкой. В дальнейшем*

при сохранении файла под текущим именем достаточно нажатия клавиши F2 (Save).

Так как Паскаль — программа MS-DOS, то и сохранять файлы рекомендуется под именами MS-DOS. Рекомендуем имя файла составлять не более чем из 8-ми латинских букв и цифр, без пробелов и знаков препинания.

4. Запуск компилятора

- Выберите меню Compile (Alt+C).
- Выберите в окне режима Compile операцию Compile.
- Нажмите клавишу Enter.

Если компилятор не обнаружил ошибок, то на экране появляется окно с сообщением: Compilation successful: press any key («Компиляция прошла успешно: нажмите любую клавишу»).

Окно остается на экране до тех пор, пока вы не нажмете какую-либо клавишу. Если обнаружена ошибка, курсор устанавливается на ошибку в окне редактирования и выдается сообщение об ошибке. Получить подробную информацию о найденной ошибке можно, нажав сразу клавишу F1. Esc — возврат в окно редактирования.

ЗАМЕЧАНИЕ

Справку по языку Паскаль можно получить, если установить курсор с помощью клавиатуры на служебное слово Паскаля и нажать комбинацию клавиш Ctrl+F1.

СОВЕТ

Быстро запустить процесс компиляции можно нажатием клавиш Alt+F9.

5. Выполнение программы

Объединяем работу построителя (команда Build) и запуск нашей программы на выполнение.

- В главном меню выберите Run (Alt+R).
- В раскрывающемся меню Run выберите команду Run,
- Нажмите клавишу Enter.

ЗАМЕЧАНИЯ

Быстро запустить программу можно нажатием комбинации клавиш Ctrl+F9.

Процессы компиляции и запуска программы на выполнение можно объединить, вызвав команду Run *сразу после набора текста программы.*

6. Просмотр результатов работы программы

- Одновременно нажмите клавиши Alt+F5 для перехода к экрану пользователя.
- Просмотрите результаты работы программы.
- Для возвращения в среду Turbo Pascal нажмите любую клавишу.
- Если результат работы вас не удовлетворяет, вернитесь к редактированию текста программы.

7. Выход из среды Turbo Pascal

- Нажмите Alt+X.

Самый простой и быстрый способ выйти из среды Turbo Pascal — нажать комбинацию клавиш Alt+X. Если ваша программа при этом не была сохранена, появится сообщение о том, что файл был изменен, и предложение его сохранить. Другой способ выйти из Turbo Pascal — выбрать в меню File команду Exit. Напоминаем, что щелчок мышью на «крестике» в правом верхнем углу окна Turbo Pascal в системе Windows является не выходом, а аварийным завершением работы. При этом набранная вами программа будет, скорее всего, потеряна.

Задание 1.2. Напишите программу, которая выводит на экран текст:

```
Важно
не путать Write
и Writeln!
```

Урок 1.3. Оформление текста на экране

До сих пор мы выводили текст шрифтом белого цвета на черном экране, начиная с той позиции, где в настоящее время находится курсор. А нельзя ли выводить текст более красиво — например, цветными буквами в центре экрана?

Для реализации такой возможности в комплект Turbo Pascal входит особый дополнительный модуль. Он называется Crt (это английская аббревиатура, обозначающая электронно-лучевую трубку — название модуля подчеркивает, что он умеет управлять способами вывода на экран).

Модуль не входит в стандарт языка, он является расширением возможностей Паскаля для IBM-совместимых компьютеров (на которых мы с вами и работаем). Этот модуль содержит набор программ (процедур), которые позволяют задавать цвет символов, очищать экран, устанавливать курсор в любую позицию экрана и выполнять множество других полезных действий.

Рассмотрим принцип работы с модулем Crt и его основные процедуры.

Как мы уже говорили, Turbo Pascal работает в текстовом режиме. Это означает, что информация на экран выводится в виде символов, каждый из которых отображается на экране в определенной позиции, как бы в клеточке. Экран при этом можно себе представить как таблицу из 25 строк и 80 столбцов (рис. 1.2). Каждая ячейка этой таблицы имеет 2 координаты — `x` и `y`, где `x` — номер столбца, `y` — номер строки. Строки нумеруются сверху вниз, начиная с единицы до 25, столбцы — слева направо, с 1-го до 80-го. То есть левый верхний угол экрана имеет координаты (1,1), правый верхний — (80,1), а левый нижний — (1,25). Символы можно выводить на экран 16 различными цветами, которые кодируются числами от 0 до 15. Каждому коду соответствует свой цвет. Полную таблицу кодов можно получить, набрав, например, слово `red` в окне редактора Turbo Pascal и нажав `Ctrl+F1` на клавиатуре.

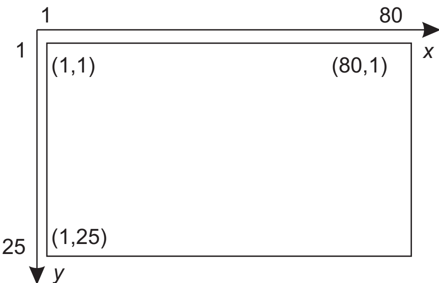


Рис. 1.2. Схема нумерации позиций экрана для модуля CRT

Внимательно разберите следующую программу:

Пример 1.3. Использование модуля Crt

```
program Second;
{ Использование возможностей модуля Crt при выводе
  на экран. Для того чтобы в программе можно было
  использовать дополнительные библиотечные функции
  (нам сейчас нужен блок функций CRT), необходимо
  в начале программы указать это в специальной секции
  объявления библиотечных модулей.
  Она начинается словом uses.
  Затем через запятую перечисляются подключаемые модули.
  Заканчивается список символом ";"}
uses Crt;               { Crt - Имя подключаемого модуля.
                          Блок заканчивается символом ";" }
{ Начало основной программы }
begin
   TextBackGround (3);   { Вызов процедуры для выбора
                          фонового цвета.
                          В скобках указан номер
                          выбранного цвета
                          В данном случае "3"
                          означает светло-голубой цвет.
                          Вместо номера можно написать
                          название цвета: black, red,
                          green, blue, magenta, cyan, … }
   ClrScr;              { Процедура очистки экрана.
                          Указав цвет фона до команды
                          ClrScr, мы тем самым залили
                          экран светло-голубым цветом }

   TextColor (14);      { Процедура выбора цвета
                          выдаваемых символов.
                          В скобках указан номер
                          выбранного цвета.
                          В данном случае желтый цвет.
                          Обратите внимание:
                          команда TextColor не меняет
                          цвет символов, уже имеющихся
                          на экране! Она лишь
                          устанавливает цвет, которым
                          будут выведены следующие
                          символы }
```

```
   GoToXY (40,10);     { Процедура установки
                         курсора в точку экрана
                         с координатами X=40, Y=10 }

   Writeln(' Все отлично!!!');
                       { Вывод текста в 10 строку,
                         начиная с позиции 40 }

   Delay (1000)        { Процедура временной задержки
                         на 1000 мкс }
                       { На современных компьютерах
                         Delay(1) обычно работает
                         быстрее, чем 1/1000 секунды.
                         Поэтому задержка в данном
                         случае будет меньше секунды }
end.                   { Конец программы }
```

Задание 1.3. Написать программу, выводящую два любых сообщения в левом верхнем и правом нижнем углах экрана. Каждое сообщение выводить своим цветом.

ЗАМЕЧАНИЕ

Прежде чем писать программу на языке программирования, стоит описать задачу словами пошагово — то есть придумать алгоритм задачи. Алгоритм можно представить в виде блок-схемы (рис. 1.3). Основные блоки, чаще всего используемые в таких схемах, см. в приложении № 1.

Алгоритм, представленный на рисунке, называется *линейным*, так как все его шаги проходятся обязательно и последовательно один за другим.

Алгоритм не зависит от языка, на котором вы программируете. Хотя в дальнейших задачах при *детализации* отдельных шагов мы будем учитывать возможности языка.

Сейчас ваша задача — представить каждый шаг (блок) алгоритма на языке Паскаль и оформить программу по образцу примера 1.3.

Задание 1.4. Написать программу, которая очищает экран и выводит слова red, green, blue, yellow каждое своим цветом в центр четвертей экрана (если экран условно разбить на 4 части, как показано на рисунке):

```
+----------------+-----------------+
|                |                 +
|     green      |      blue       +
|                |                 +
+----------------+-----------------+
|                |                 +
|    yellow      |       red       +
|                |                 +
+----------------+-----------------+
```

Задание 1.5. Левый столбец таблицы содержит действия, которые выполняет некоторый оператор. Правый столбец содержит операторы языка Паскаль. Поставьте в соответствие элементам из левого столбца таблицы элементы из правого столбца.

1. Очистка экрана	A. `Crt;`
2. Позиционирование курсора в левый нижний угол экрана	B. `TextBackGround(red); ClrScr;`
3. Заказ красного цвета фона экрана	C. `Write('Happy New Year');`
4. Заливка экрана красным цветом	D. `GoToXY(78,1);`
5. Вывод в текущую позицию экрана 'Happy New Year' с переходом курсора на новую строку	E. `TextColor(red);`
6. Позиционирование курсора в правый верхний угол экрана	F. `TextColor(12); Write('Hello');`
7. Установка красного цвета текста	G. `GoToXY(1,23);`
8. Вывод в текущую позицию экрана 'Happy New Year' без перехода курсора на новую строку	H. `Writeln('Happy New Year');`
9. Библиотека среды Turbo Pascal для работы в текстовом режиме	I. `Begin end`
10. Начало и конец тела программы	J. `TextBackGround(red);`
11. Вывод в центр экрана 'Hello'	K. `ClrScr;`
12. Вывод текста 'Hello' цветом № 12	L. `GotoXY(35,12); Write('Hello');`

Ответ: 1–K, 2–G, 3–J, 4–B, 5–H, 6–D, 7–E, 8–C, 9–A, 10–I, 11–L, 12–F.

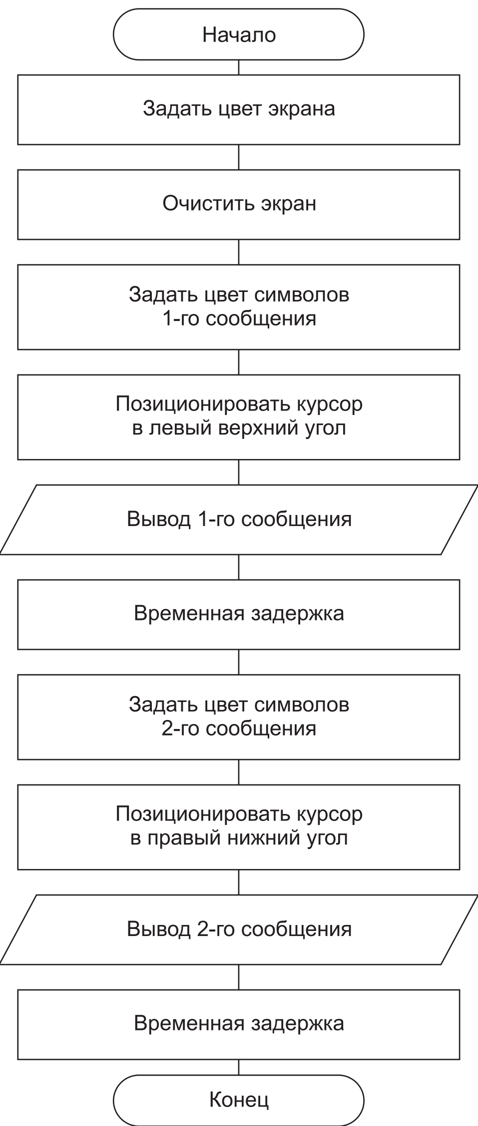


Рис. 1.3. Алгоритм вывода цветных сообщений в левом верхнем и правом нижнем углах экран

ЗАМЕЧАНИЕ

Возможно, вы обратили внимание, что мы иногда используем термин «Паскаль», а в других местах «Trubo Pascal». Этим мы обозначаем разницу между языком программирования Паскаль и средой программирования Turbo Pascal.

Выводы

1. Любую задачу можно представить в виде последовательности шагов — алгоритма. Одна из форм записи алгоритма — блок-схема, которая в дальнейшем переводится на конкретный язык программирования.
2. В структуре программы на языке программирования Паскаль обязательно присутствует тело программы. Его формируют операторы `Begin` и `End`. Между `Begin` и `End` с помощью других операторов задаются определенные действия.
3. Вывод информации на экран осуществляют операторы `write` и `writeln`.
4. При выполнении некоторых действий в Turbo-среде используются библиотечные модули языка Паскаль. Имена этих модулей объявляются в разделе `uses`.
5. Для красивого вывода на экран используется модуль Crt. Он позволяет очищать экран (`ClrScr`), менять позицию курсора (`GotoXY`), а также цвет символов (`TextColor`) и фона (`TextBack-Ground`).

Контрольные вопросы

1. Какими словами начинается и заканчивается тело любой программы на Паскале?
2. Из каких этапов состоит процесс создания компьютерной программы?
3. Как запустить Turbo Pascal? Как выйти из среды Turbo Pascal? Как запустить программу в среде Turbo Pascal?
4. Какие действия совершают операторы `write` и `writeln`? В чем состоит разница между ними?

5. Какие действия нужно совершить, чтобы скопировать часть программы в другое место? Какие комбинации клавиш при этом нужно использовать?
6. Как просмотреть результат работы программы?
7. Что такое текстовый режим работы? Опишите правило задания координат определенной точки экрана.
8. Какой модуль позволяет выводить информацию на экран красиво и в цвете? Как установить курсор в нужную позицию экрана?
9. Что такое линейный алгоритм?

ТЕМА 2

Как включить в работу числовые данные

Мы рассмотрели простейшие действия — вывод на экран информации, причем всегда одной и той же. Однако компьютеры были придуманы для автоматизации сложных вычислений, для быстрого выполнения математических операций (иначе говоря — для работы с числами). И до сих пор эта их функция остается главной: ведь вся компьютерная информация хранится в форме чисел. Цель нашей второй темы — научиться работать с различными видами чисел.

Урок 2.1. Начнем с простого: целые числа

Любые данные, с которыми вы работаете, необходимо где-то хранить. Все данные, с которыми работает программа, должны находиться в основной памяти. Основная память состоит из ячеек (байтов), каждая из которых имеет *адрес*, то есть порядковый номер (рис. 2.1). В этих ячейках мы и будем хранить данные.

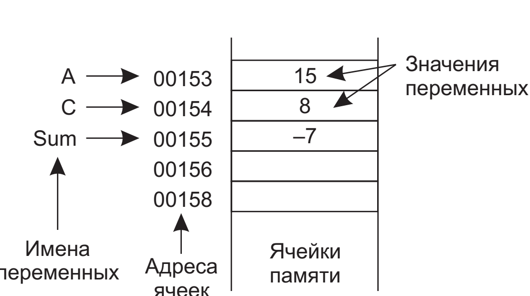


Рис. 2.1. Хранение переменных в памяти компьютера

Понятие переменной

Данные нашей программы принято называть *величинами*. Величины, которые меняются, называют *переменными*, а те, которые не меняются — *постоянными*.

Величину (число), хранящуюся в ячейке, называют *значением ячейки*. Программа работает с адресами и значениями ячеек памяти. Но нам, людям, неудобно работать с адресами — это большие числа и они для нас ничего не значат. Поэтому ячейкам, с которыми будет работать программа, принято давать *имя*, или, что то же самое, *идентификатор*. В специальной таблице программа-компилятор будет запоминать, какому имени какой адрес ячейки памяти соответствует. Итак, мы будем иметь дело только с именами ячеек и с их значениями.

ЗАПОМНИТЕ!

Идентификатор (имя) всегда должен начинаться с латинской буквы, после которой может следовать некоторое число латинских букв, цифр или символов подчеркивания (_). В имени не должно быть пробелов, запятых или других непредусмотренных знаков. В Turbo Pascal 7.1 учитываются лишь первые 63 символа.

Тип Integer. Оператор присваивания. Вывод на экран

Теперь рассмотрим работу с самыми простыми переменными. В них можно хранить только целые числа. Для хранения целых чисел в Паскале используется специальный тип данных — `Integer`.

Внимательно читайте комментарии к программе в следующем примере!

Пример 2.1. Работа с целочисленными переменными

```
Program Product;

{ Далее идет раздел описания переменных. Он всегда
  начинается со слова var (от variable — переменная) }
var

   A,B,C: integer;   { Имена в списке — через запятую;
                       в конце списка через двоеточие
```

```
                       указывается тип данных:
                       integer – целый }

Begin                { Началось тело программы }

   A:=5;             { Это оператор присваивания.
                       В данном случае запись означает,
                       что в переменную (ячейку) A
                       записали число 5.
                       Не путайте с записью A=5 !!! }

   writeln(A);       { Выводим на экран содержимое
                       переменной A.
                       Имя A не заключено в апострофы! }

   writeln('A');     { Вывод на экран символа A }

   A:=A+1;           { Запишем в переменную A число,
                       которое до этого в ней было,
                       но увеличенное на 1 }
   B:=7;

   C:=A*B;           { * – это операция умножения }

   writeln('Product=',C)  { Вывод содержимого ячейки C
                             с пояснительным текстом }

end.                 { Здесь кончается тело программы }
```

При запуске программа выведет на экран следующее:

```
5
A
Product=42
```

ЗАПОМНИТЕ!

В результате выполнения оператора присваивания в ячейку помещается новое число. Старое содержимое ячейки при этом пропадает.

Справа от оператора присваивания может стоять число или любое выражение. Слева может стоять только имя переменной. Выражения слева быть не может — иначе Паскаль не будет знать, в какую ячейку памяти поместить результат.

Тип результата выражения справа от оператора присваивания должен быть таким, чтобы помещаться в переменную слева от «:=».

Операции с типом Integer

Рассмотрим операции, которые можно выполнять с целыми числами и целочисленными переменными.

Пример 2.2. Операции с переменными целого типа

```
Program Action;
var
   A,B,C: integer;
begin
   A:=17;
   B:=3;
   { Операция умножения: }
   C:=A*B;     writeln('17 * 3=',C);

   { Деление нацело: }
   C:=A div B; writeln('17 div 3=',C);

   { Вычисление остатка от деления: }
   C:=A mod B; writeln('17 mod 3=',C);

   { Сложение: }
   C:=A+B;     writeln('17 + 3=',C);

   { Вычитание: }
   C:=A-B;     writeln('17 - 3=',C)
end.
```

При запуске программа выведет на экран следующее:

```
17 * 3=51
17 div 3=5
17 mod 3=2
17 + 3=20
17 - 3=14
```

Рассмотрим еще несколько примеров операций div и mod. Для успешного понимания результатов этих операций нужно вспомнить 2-й класс и деление столбиком (рис. 2.2).

```
_ 23 | 5       23 div 5 = 4
  20 |----     23 mod 5 = 3
  -- | 4
   3
```

Рис. 2.2. Пример целочисленного деления столбиком

Частая ошибка: не забудьте, что все действия мы производим только с целыми числами! Не нужно продолжать деление, когда делимое (это то, что мы делим) оказывается меньше делителя (это то, на что мы делим). То, что осталось от делимого, называется *остатком*. Это и есть результат операции mod. Целое число, которое получилось в результате деления, называется *целочисленным частным*. Это результат операции div.

Проверим себя, вспомнив 2-й класс:

```
 5 div 2=2;  5 mod 2=1;
 6 div 2=3;  6 mod 2=0;
40 div 6=5; 40 mod 6=4;
 3 div 5=0;  3 mod 5=3.
```

Результат вычисления операций div и mod для отрицательных чисел оказывается не совсем таким, как положено в математике (когда остаток всегда неотрицателен). Зато он более понятен. Другими словами, результат нужно посчитать отдельно от знаков, а потом добавить знак в соответствии с правилами математики:

```
(–10) div 3=–3; (–10) mod 3=–1;
(–3) div 5=0; (–3) mod 5=–3.
```

Задание 2.1. Даны 3 целых числа *A*, *B*, *C*. Вычислить их сумму и произведение.

Продумаем алгоритм решения данной задачи (рис. 2.3).

Представьте каждый шаг алгоритма на языке Паскаль.

Задание 2.2. Дана длина ребра куба (целое число). Найти объем куба и площадь его боковой поверхности.

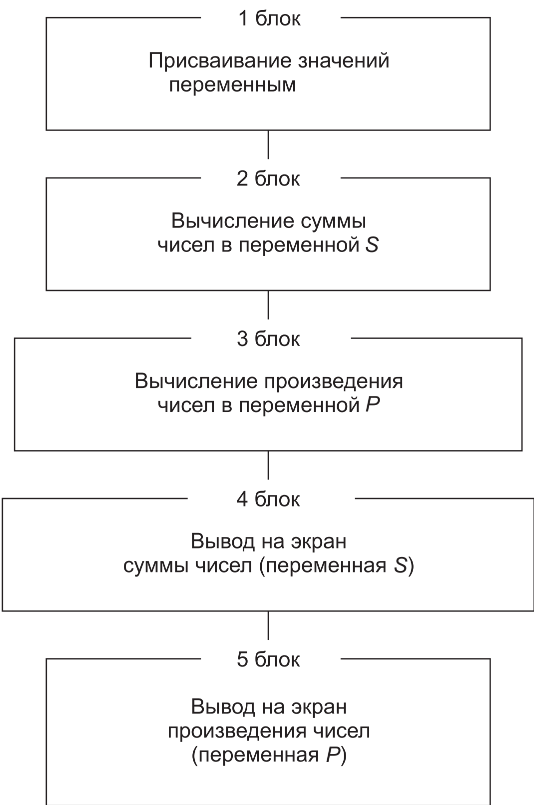


Рис. 2.3. Блок-схема алгоритма решения задания 2.1

Стандартные функции типа Integer

Многие стандартные действия с числовыми данными выполняются путем вызова функций из библиотеки Паскаля. Такие функции называются *стандартными функциями*.

Пример 2.3. Демонстрация стандартных функций

```
Program Infunct;
var
   A,B,C: integer;
begin
   A:=-2;
```

```
    { Функция Abs (X) вычисляет абсолютное значение
      аргумента X, то есть модуль X }
    B:=Abs(A); writeln('Abs(-2)=',B);

    { Функция Sqr (X) возводит в квадрат аргумент X }
    C:=Sqr(B); writeln('Sqr(B)=',C);
    C:=Sqr(B+B); writeln('Sqr(B+B)=',C)

end.
```

При запуске программы вывод на экран:

```
Abs(-2)=2
Sqr(B)=4
Sqr(B+B)=16
```

Задание 2.3. Вычислите значение выражения $|39 \cdot 54 - 84^2|$.

Задание 2.4. В переменные A и B записаны целые числа (оператором присваивания, например, A:=20; B:=13). Поменяйте числа в этих переменных местами.

Будьте внимательны! Если записать A:=B, вы потеряете число 20 и получите в двух переменных число 13! Воспользуйтесь третьей переменной — C (рис. 2.4).

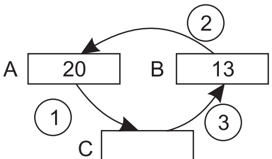


Рис. 2.4. Схема обмена значений двух переменных через третью ячейку. В кружках указан порядок операторов присваивания

Задание 2.5. Выполните задание 4 без использования третьей переменной. Используйте действия сложения и вычитания.

Как представляются переменные целого типа в памяти компьютера

Вся информация в компьютере хранится в виде последовательностей нулей и единиц. Информация, для записи которой исполь-

зуется всего два знака: 0 и 1, называется *двоичной*. Информация в компьютере хранится в виде двоичных кодов (комбинации из нулей и единиц). Память мы представляем, как последовательность ячеек, каждая из которых имеет свой адрес (см. рис. 2.1).

Стандартная длина ячейки — 8 бит, что равно 1 байт. В такую ячейку можно записать двоичный код длиной 8 бит.

Для переменной типа `integer` выделяется ячейка длиной в 2 байта = 16 бит. Такая ячейка получает символьное имя — имя переменной, и вы обращаетесь к ней не по адресу, а по имени.

Крайняя левая позиция выделяется для знака числа:

- 0 — число положительно;
- 1 — число отрицательно.

Остальные 15 позиций выделяются для записи самого числа в двоичном виде (рис. 2.5).

Рис. 2.5. Распределение двоичных разрядов (бит) при хранении числа типа integer

На 15 позициях можно получить 2^{15} двоичных кодов. Самое маленькое число состоит из 15 нулей, самое большое — из 15 единиц. Поскольку счет начинается с нуля, получаем всего $(2^{15} - 1)$ положительных двоичных чисел. С учетом знака числа (+ или –) получаем, что числа типа `integer` имеют диапазон представления $-2^{15}...+2^{15} - 1$, или –32 768 ... +32 767.

Тип integer является основой для нескольких производных типов — со знаком и без знака (табл. 2.1).

Задание 2.6. Считая, что операция умножения и операция возведения в квадрат имеют одинаковую сложность, запишите оптимальным образом выражения:

а) x^5;
б) x^6;
в) x^8;
г) x^9;
д) x^{10}.

Таблица 2.1. Целочисленные типы данных среды Turbo Pascal

Наличие знака	Тип переменной	Формат (длина) в байтах	Диапазон	
			Запись с порядком	Обычная запись
Без знака	byte	1	$0 \ldots 2^8 - 1$	0 ... 255
	word	2	$0 \ldots 2^{16} - 1$	0 ... 65 535
Со знаком	shortint	1	$-2^7 \ldots 2^7 - 1$	−128 ... 127
	integer	2	$-2^{15} \ldots 2^{15} - 1$	−32 768 ... 32 767
	longint	4	$-2^{31} \ldots 2^{31} - 1$	−2 147 483 648 ... 2 147 483 647

Урок 2.2. Включаем в работу вещественные числа

Что делать, если число не целое, то есть имеет десятичную точку?

Описание вещественного типа данных (real)

Опишем в программе переменные для хранения не целых чисел. Такие переменные имеют тип real — вещественный тип.

Пример 2.4. Работа с типом real

```
Program Optel;
var
   A,B,C: real;
begin
   A:=3.5;
   B:=7.6;
   C:=A+B;
   writeln('Сумма=',C)
end.
```

При запуске программы вывод на экран:

```
Сумма=1.1100000000E+01
```

Форматы записи вещественных переменных

В примере 2.4 переменная вещественного типа будет выдана на экран в особой, *экспоненциальной* форме. Числа с десятичной точкой могут записываться в двух формах:

1. Обычная форма.

   Примеры:

- 0,7 может быть записано как `0.7` или `.7`
- –2,1 может быть записано как `–2.1`

2. Запись с экспонентой: число представляется в виде мантиссы, то есть дробной части числа, умноженной на 10 в некоторой степени.

   Примеры:

- $2700 = 2{,}7 \cdot 10^3$. Число 10 записывается в виде буквы E, а за ней идет величина степени: `2.7E3`;
- $0{,}002 = 2 \cdot 10^{-3}$ соответствует запись `2E–3`.

Вещественные операции

Пример 2.5. Операции с переменными вещественного типа

```
program Operation;
var
   A,B,C: real;
begin
   A:=17.3;
   B:=3.4;
   C:=A*B; writeln('A*B=',C);

   { / — это операция деления }
   C:=A/B; writeln('A/B=',C);

   C:=A+B; writeln('A+B=',C);
   C:=A-B; writeln('A-B=',C)
end.
```

При запуске программа выведет на экран следующее:

```
A*B= 5.8820000000E+01
A/B= 5.0882352941E+00
```

```
A+B= 2.0700000000E+01
A-B= 1.3900000000E+01
```

ЗАМАЧАНИЕ

Для вещественных чисел нет таких проблем с операцией деления, как для целых чисел. Операция «/» — это обычное деление.

Задание 2.7. Вычислите выражение:

$$\frac{7.478937 - 89.2456}{883.5995 + 618.332} \times 76.2833.$$

Стандартные функции типа real

Пример 2.6. Стандартные функции с вещественными переменными

```
Program AllFunc;
var
   A,B: real;
begin
   A:=2.0;
   B:=Sqr(A); writeln('Sqr(2.0)=',B);
   B:=Abs(-A); writeln('Abs(-2.0)=',B);

   B:=Sqrt (A);      { Вычисление квадратного корня }
   writeln('Sqrt(2)=',B);

   B:=Sin (A);       { Вычисление синуса }

   { Зададим вывод вещественного числа
     не в экспоненциальной, а в обычной форме. На экране
     под значение переменной "B" закажем 6 позиций. Из
     них 3 позиции выделим для цифр справа от десятичной
     точки }
   writeln('Sin(2)=',B:6:3);

   B:=Cos (A);       { Вычисление косинуса }
   writeln('Cos(2)=',B:6:3);

   B:=Arctan (A);    { Вычисление арктангенса }
   writeln('Arctan(2)=',B);
   B:=Ln (A);        { Вычисление логарифма }
```

```
   writeln('Ln(2)=',B);

   B:=Exp (A);      { Возведение числа e в степень A }
   writeln('Exp(2)=',B);
   B:=Pi;          { Вычисление числа Пи }
   writeln('Pi=',B)
end.
```

При запуске программы выведет на экран следующее:

```
Sqr(2.0)= 4.0000000000E+00
Abs(-2.0)= 2.0000000000E+00
Sqrt(2)= 1.4142135624E+00
Sin(2)= 0.909
Cos(2)=-0.416
Arctan(2)= 1.1071487178E+00
Ln(2)= 6.9314718056E-01
Exp(2)= 7.3890560989E+00
Pi= 3.1415926536E+00
```

ЗАМЕЧАНИЕ

Вы, вероятно, заметили, что Паскаль содержит мало функций. Он не умеет вычислять даже те функции, которые вычисляет обычный инженерный калькулятор! Что же это за язык программирования, скажете вы! Ответ на это прост: Паскаль разрабатывался не для вычислений (как, например, Фортран), а для обучения.

Запись математических выражений

Имеющихся в Паскале функций достаточно для вычисления других, более сложных. Вот несколько примеров:

$$\operatorname{tg} x = \frac{\sin x}{\cos x}, \quad \operatorname{ctg} x = \frac{\cos x}{\sin x}, \quad \log_y x = \frac{\ln x}{\ln y}, \quad x^y = e^{y \ln x}.$$

Например, чтобы вычислить $(2x+3)^{1+\cos x}$, мы напишем на Паскале:

```
exp( (1+cos(x)) * ln(2*x+3) )
```

Обратите внимание на то, что при записи выражений на языке Паскаль нужно тщательно задумываться о приоритетах операций. Например, выражение $\frac{x+1}{2x}$, записанное в виде x+1/2x, содержит сразу три ошибки. Во-первых, приоритет операции деления выше, чем у сложения, поэтому, для правильного вычисления числителя его надо взять в скобки: (x+1). Во-вторых, Паскаль не понимает, что означает 2x. Это мы привыкли, что в математике операцию умножения в таких случаях опускают. Паскалю требуется, чтобы она была указана явно: (x+1)/2*x. Но даже это выражение все еще содержит ошибку. Дело в том, что умножение и деление имеют одинаковый приоритет и выполняются слева направо. Значит, при такой записи сначала выполнится деление, а потом результат будет умножен на x. Нужно либо поставить знаменатель в скобки и написать (x+1)/(2*x), либо, для ленивых, поставить вместо умножения деление: (x+1)/2/x. Порядок вычисления в этом случае будет не такой, как требует условие. Однако результат будет таким же: ведь поделить на $2x$ — это все равно, что поделить на 2, а потом результат — на x!

ЗАПОМНИТЕ!

Аргументы функции всегда пишутся в скобках. То есть если у функции нет аргументов (как у Pi, например), то скобки после ее имени не нужны. Если же аргументы есть, то после имени функции вы должны обязательно открыть скобку, перечислить аргументы и не забыть закрыть скобку. Например, sin 2x *в Паскале нужно записывать как* sin(2*x).

Задание 2.8. Напишите программу для вычисления дискриминанты квадратного уравнения. Коэффициенты задайте в программе через оператор присваивания.

Продумаем алгоритм решения данной задачи (рис. 2.6.).

Запишите каждый шаг алгоритма на языке Паскаль.

Задание 2.9. Вычислите выражение:

$$\sqrt{\operatorname{arctg}^2\left(\sin\frac{3,15}{6,1}\right) + 53,7}\ .$$

Рис. 2.6. Алгоритм выполнения задания 2.8

Как представляются переменные вещественного типа в памяти компьютера

Вы познакомились с экспоненциальной формой представления числа. В ней можно выделить две части: мантиссу, то есть значащие цифры числа, и порядок — степень десятки (в общем случае это степень основания системы счисления, в которой записано данное число). Ячейка памяти, выделенная для переменной вещественного типа, должна содержать следующие элементы: знак числа, знак порядка, значение порядка и значение мантиссы — естественно, все в двоичном представлении (рис. 2.7)

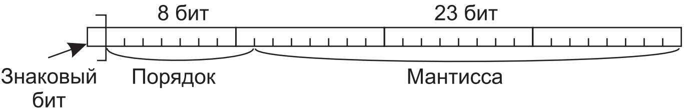


Рис. 2.7. Распределение двоичных разрядов (бит) при хранении числа типа real

Порядок и знак порядка занимают вместе 8 бит и хранятся, вообще говоря, немного хитрее. Но это нам сейчас не важно. Главное — понять принцип хранения чисел с плавающей точкой.

Таблица 2.2. Вещественные типы данных среды Turbo Pascal

Тип переменной	Формат (длина) в байтах	Примерный диапазон абсолютных значений	Количество значащих десятичных цифр
Одинарный (single)	4	10^{-45} ... 10^{38}	7 или 8
Вещественный (real)	6	10^{-39} ... 10^{38}	11 или 12
Двойной (double)	8	10^{-324} ... 10^{308}	15 или 16
Расширенный (extended)	10	10^{-4932} ... 10^{4932}	19 или 20

Урок 2.3. Как совместить переменные целого и вещественного типа

В программе могут одновременно встречаться переменные разных типов. Как их совместить?

Преобразование типов

Пример 2.7. Одновременное использование вещественных и целых типов

```
Program Mix;
var
   N,K: integer;
   A,B: real;
begin
   N:=4;
   A:=3.6;

   B:=N;            { В переменную типа real можно
                      записать целое число }

   writeln('B=',B);

   { В переменную типа integer нельзя записать
     вещественное число! Чтобы все-таки поместить число
     типа real в переменную типа integer, нужно явно
```

```
    указать, что делать с дробной частью числа. Есть
    два варианта: }

  N:=Trunc(A);     { Функция Trunc(X) возвращает
                     целую часть числа X,
                     то есть, отбрасывает дробную часть }
  writeln('Trunc(3.6)=',N);

  K:=Round(A);     { Функция Round(X) — округляет
                     до ближайшего целого }

  writeln('Round(3.6)=',K)
end.
```

При запуске программа выведет на экран следующее:

```
B:= 4.0000000000E+00
Trunc(3.6)=3
Round(3.6)=4
```

Еще раз уточним правила преобразования типов: для хранения данных типа `integer` используется 2 байта, а для `real` необходимо 6 байт. Это значит, что число типа `integer` можно поместить в ячейку типа `real` (целая часть будет равна этому числу, а дробная будет равна нулю). А вот число типа `real` в ячейку типа `integer` никак не поместится. Чтобы все-таки поместить его туда, нужно явно указать, что делать с дробной частью числа. Для этого предусмотрены 2 функции: `trunc` и `round`. Обе они возвращают результат типа `integer`.

Что делать, если в программе нужно записать сложное математическое выражение? В каком порядке будут выполняться действия?

Правила приоритета в выполняемых действиях

1. Действия над переменными, стоящими в скобках, выполняются в первую очередь.
2. После вычисления значений всех скобок вычисляются все функции.
3. После функций выполняются умножение и деление. Они имеют одинаковый приоритет.

4. Следующие по приоритету — сложение и вычитание.
5. Операции одинакового приоритета выполняются слева направо.

Действия над данными разных типов

Сведем воедино операции и функции по работе с вещественными и целыми величинами.

Таблица 2.3. Операции и функции для типов integer и real

Операция/функция	Тип данных 1-го аргумента	Тип данных 2-го аргумента	Тип результата
+, -, *	Integer	Integer	Integer
	Integer	Real	Real
	Real	Integer	Real
	Real	Real	Real
/	Не важен		Real
Div, mod	Только Integer		Integer
Abs, Sqr	Integer	—	Integer
	Real	—	Real
Sqrt, Sin, Cos, Arctan, Ln, Exp, Pi	Не важен	—	Real
Trunc, Round	Не важен	—	Integer

Поясним написанное в таблице. Мы разделили все функции/операции на 6 категорий.

1. Результат операций +, – и * зависит от типа аргументов. Если хоть один из них имеет тип real, то и результат будет иметь тип real. Это объясняется тем, что у данных типа real есть дробная часть, а у integer — нет. Даже если в вещественной переменной хранится целое число, оно все равно имеет дробную часть, только она равна нулю. То есть, если хотя бы у одного из аргументов есть дробная часть, то в результате выполнения операции она никуда не исчезает. Поэтому результат тоже имеет дробную часть (real).

2. Результат операции вещественного деления по определению всегда имеет дробную часть.
3. Операции целочисленного деления определены только для целых чисел. Поэтому результат тоже всегда целый.
4. Функции Abs и Sqr определены для обоих типов данных. Поэтому тип их результата зависит от типа аргумента. Для целого аргумента результат имеет целый тип, для вещественного — вещественный.
5. Функции Sqrt, Sin, Cos, Arctan, Ln, Exp, Pi по определению являются вещественными. (На самом деле это связано с особенностями вычисления Паскалем этих функций. Он вычисляет их приближенно путем разложения в ряд. Такой метод не предполагает целого результата. Более того, значения этих функций всегда вычисляются приближенно.)
6. Функции Trunc и Round предназначены для преобразования типов. Они явно указывают на то, что сделать с дробной частью числа. Поэтому это единственный способ получить на Паскале из дробного числа целое.

Изложенные выше сведения позволяют нам понимать, что за выражение написано в чужой программе, какое оно будет иметь значение, и какого типа будет результат.

Задание 2.10. Вычислите выражение и укажите тип результата:

```
Abs(12 mod 7*4/2-350 div 15)+2
```

Сначала расставим операции в соответствии с их приоритетами (табл. 2.4).

Таблица 2.4. Расстановка операций в примере 2.8

№ пп	Операция	Пояснение
1	12 mod 7	Сначала выполняются действия в скобках. Скобки в данном случае ограничивают аргумент функции Abs. В скобках сначала выполняются операции типа «умножение-деление», а потом — «сложение-вычитание». Операции mod и div осуществляют целочисленное деление. Их приоритет такой же, как у операций «*» и «/». В первой группе таких операций три. Выполняем их слева направо. Поэтому сначала выполняем mod, затем будет «*», а потом «/»

№ пп	Операция	Пояснение
2	(12 mod 7) * 4	
3	(12 mod 7*4) / 2	
4	350 div 15	Теперь выполняем вторую группу операций типа «умножение-деление». В данном случае это одинокая операция div
5	(12 mod 7*4/2) - (350 div 15)	Теперь пришла пора объединить результаты первых двух групп операцией «–». Она выполняется последней в скобках, так как имеет наименьший приоритет
6	Abs(…)	Теперь пора посчитать результат функции
7	Abs(…) + 2	Последний оператор — сложение

Нам представляется удобным расставлять приоритеты, обводя операции (рис. 2.8). Так оказывается легче понять, что уже вычислено и что еще предстоит вычислить.

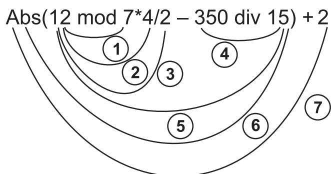


Рис. 2.8. Порядок выполнения операций в задании 2.10

Теперь определим тип и результаты каждого действия (табл. 2.5).

Таблица 2.5. Вычисление результатов каждого действия из задания 2.10

№ пп	Операция	Результат и его тип	Пояснение
1	12 mod 7	5 Integer	Остаток от деления 12 на 7 равен 5. Результат операции mod всегда целый
2	5 * 4	20 Integer	Результат умножения двух целых — целое число

продолжение ⇨

Таблица 2.5 (*продолжение*)

№ пп	Операция	Результат и его тип	Пояснение
3	`20 / 2`	`10.0` `Real`	Результат вещественного деления всегда вещественный. Хотя 20 делится нацело на 2, мы специально приписали «.0» к результату, чтобы подчеркнуть и не забыть, что результат вещественный и имеет дробную часть
4	`350 div 15`	`23` `Integer`	При целочисленном делении нас интересует только целая часть частного. Результат операции div всегда целый
5	`10.0 - 23`	`-13.0` `Real`	При вычитании одно из чисел имеет дробную часть. Команды от нее избавиться не было, поэтому результат тоже имеет дробную часть. Значит, результат имеет тип `Real`
6	`Abs(-13.0)`	`13.0` `Real`	Abs меняет отрицательный знак аргумента на положительный. Так как аргумент имеет дробную часть, а команды от нее избавиться не было, результат тоже содержит дробную часть
7	`13.0 + 2`	`15.0` `Real`	Те же соображения, что и в 5-м пункте. Результат имеет дробную часть
Ответ: 15.0. Real			

Задание 2.11. Дано действительное число *X*. Напишите программу для вычисления:

- целой части числа *X*;
- числа *X*, округленного до ближайшего целого;
- числа *X* без дробных цифр.

Урок 2.4. Ввод и вывод данных

Заносить данные в ячейки памяти можно не только оператором присваивания, но и путем непосредственного ввода с клавиату-

ры. Это удобно тем, что в программу при каждом запуске можно вводить разные начальные значения, что добавляет ей универсальности.

Вводим переменные с клавиатуры

Пример 2.8. Ввод с клавиатуры значения переменной N

```
program Inp;
uses Crt;
var
   N: integer;
begin
   ClrScr;
   write('Введите число с клавиатуры:');

   readln(N);     { Здесь программа приостановится
                    и будет ожидать ввод с клавиатуры.
                    Наберите на клавиатуре число,
                    например 153, и нажмите
                    клавишу Enter}

   writeln('Вы ввели число ', N);

   readln         { Это оператор пустого ввода. Здесь
                    программа опять приостановится и
                    будет ожидать нажатия клавиши Enter.
                    За это время вы успеете просмотреть
                    вывод на экране. Этот прием мы
                    рекомендуем использовать, чтобы не
                    нажимать Alt+F5 после окончания
                    работы программы }
end.
```

Красивый вывод на экран

Рассмотрим еще одну задачу: задать с клавиатуры цвет фона (экрана), символов и координат для вывода текста, а затем вывести текст в окно с заданными координатами.

Продумаем алгоритм решения данной задачи (рис. 2.9).

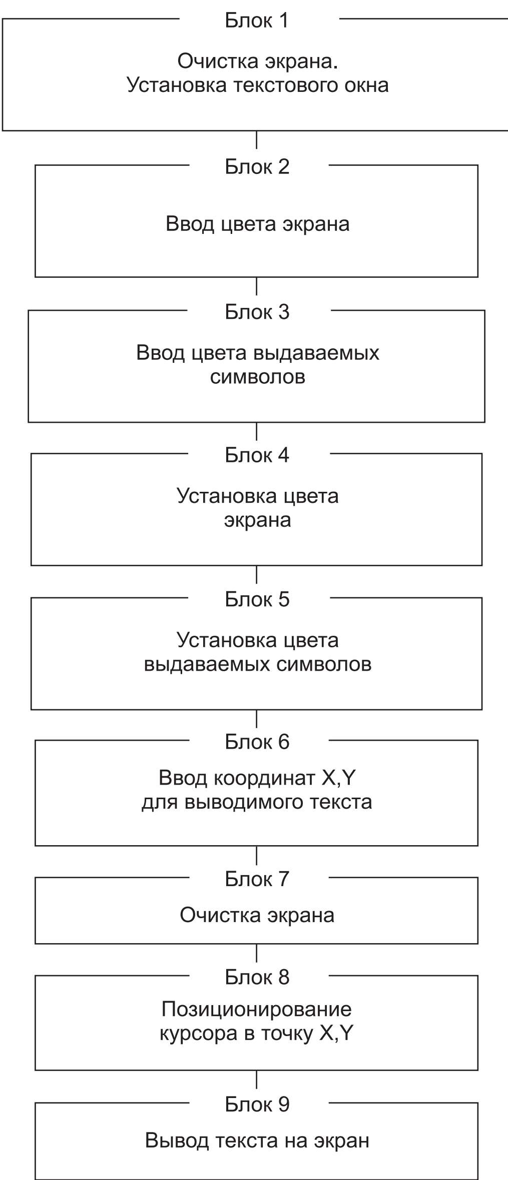


Рис. 2.9. Алгоритм решения задачи примера 2.9

Пример 2.9. Красивый вывод текста

```
program Inp_Color;
uses
   Crt;        { Подключение библиотечного модуля Crt }
var

  { Опишем переменные, где будут храниться
    цвет экрана – Cl1 и цвет выдаваемых символов – Cl2 }
  Cl1,Cl2: 0..15;   { 0..15 – этот тип для переменных
                      называется интервальным. В данном
                      случае значения переменных могут
                      меняться в интервале от 0 до 15.
                      Здесь мы имеем отрезок (интервал)
                      базового типа integer. Палитра
                      цветов лежит именно в этом
                      интервале }

   X,Y: integer;     { Для хранения координат }

begin

   { Блок 1: }
   ClrScr;

   { Ниже следует вызов процедуры Window(X1,Y1,X2,Y2) из
     модуля Crt, которая определяет на экране текстовое
     окно. (X1,Y1) – координаты верхнего левого угла
     окна, (X2,Y2) – координаты нижнего правого угла }
   Window(20,5,60,20);

   { Блок 2: }
   write('Введите цвет для экрана:');

   readln(Cl1);      { Здесь работа программы
                      приостанавливается и ожидается
                      ввод номера цвета экрана
                      в переменную Cl1. Во время работы
                      программы следует ввести число и
                      нажать Enter }

   { Блок 3: }
   write('Введите цвет для символов:');
```

```
  readln(Cl2);      { Ожидается ввод номера цвета
                     символов в переменную Cl2. Во время
                     работы программы следует ввести
                     число и нажать Enter }

  { Блок 4: }
  TextBackGround(Cl1);      { Выбор фонового цвета }

  { Блок 5: }
  TextColor(Cl2); { Выбор цвета выводимых символов }

  { Блок 6: }
  writeln('Введите координаты X и Y');
  readln(X,Y);     { Ожидается ввод координат для
                     выводимого текста. Необходимо
                     ввести два числа (координаты) через
                     пробел и нажать Enter.
                     Помните, что
                     координаты внутри текстового окна
                     отсчитываются от его левого
                     верхнего угла. В нашем окне
                     16 строк и 41 столбец}

  { Блок 7: }
  ClrScr;         { Функция очистки экрана в данном
                    случае очистит не весь экран, а
                    только заданное текстовое окно }

  { Блок 8: }
  GoToXY(X,Y);{ Позиционирование курсора в точку
                с координатами X,Y }

  { Блок 9: }
  writeln(' Мы отлично вводим с клавиатуры!');

  readln      { Этот "пустой" оператор readln
                задерживает нас в экране пользователя.
                Возврат в среду Turbo Pascal происходит
                после нажатия Enter }
end.
```

Задание значений переменных датчиком случайных чисел

Есть еще один способ занесения данных в переменные — вызов датчика случайных чисел, когда компьютер сам выдает случайное число из указанного диапазона.

Зададим этим способом цвет экрана. Вместо второго блока в программе поставим вызов датчика случайных чисел Random(X). Но перед этим обязательно нужно выполнить процедуру инициализации датчика, иначе он будет выдавать нам неслучайные числа:

```
Randomize;      { Инициализация датчика случайных чисел
                  проводится один раз в программе }

Cl1:=Random(16);{ В результате переменной Cl1
                  присваивается целое случайное число
                  из диапазона 0..15.
                  Результат функции Random(N)
                  случайное число из диапазона 0..N-1 }
```

ЗАМЕЧАНИЕ

На самом деле настоящий программный датчик случайных чисел создать невозможно. Функция Random *выдает псевдослучайные числа. Это значит, что числа, выдаваемые функцией, порождаются по определенной закономерности. Эта закономерность придумана так, чтобы казалось, что числа получаются случайными. Однако если не использовать процедуру* randomize*, каждый раз при запуске программы последовательность чисел будет одинаковой. Процедура* randomize *«встряхивает» начальное значение последовательности. После этого порождаемая последовательность чисел становится почти уникальной.*

Рассмотрим еще несколько примеров использования функции Random.

Пусть нам необходимо получить случайное двузначное число, то есть число от 10 до 99. Логика проста: нам нужно одно случайное число из 90 (именно столько всего двузначных чисел). Значит, будем использовать функцию Random, которая выдаст одно число из 90, то есть Random(90). Но результат тогда будет лежать в диапазоне 0..89, а нам требуется 10..99! Значит, нужно сдвинуть полученный диапазон вправо на 10. Это делает операция +10. В итоге получаем следующее: Random(90)+10.

Задание 2.12. Напишите программу, которая выдает сообщение в текстовое окно. Координаты окна и координаты для сообщения должны вводиться с клавиатуры. Цвет экрана и цвет символов задайте с помощью датчика случайных чисел.

Задание 2.13. Напишите программу, которая выдает сообщение на полный экран (без текстового окна). Координаты сообщения, цвет экрана и цвет символов задайте с помощью датчика случайных чисел.

ЗАПОМНИТЕ!

Заливка экрана или текстового окна выполняется вызовом двух процедур: установка цвета экрана — процедурой `TextBackGround(...)`*, а очистка экрана — процедурой* `ClrScr`*.*

Урок 2.5. Зачем нужны константы в программе?

Бывают случаи, когда некоторые величины не меняются по ходу выполнения программы. Для удобства работы с такими величинами в Паскале предусмотрена отдельная категория — константы.

Для начала простой пример:

Пример 2.10. Расчет скорости тела при падении с башни

```
Program Piza;
var
   G,V,H: real;
begin
   G:=9.8;  { Эта переменная всегда имеет одно значение
              и не изменяет его
              по ходу выполнения программы }

   write('Введите высоту башни:');
   readln(H); V:=Sqrt(2*G*H);

   writeln('Скорость падения':20,V:7:3);
           { На выводимый текст выделяется 20 позиций }

   readln
end.
```

Для наглядности в формуле вычисления скорости падения мы использовали переменную G, которая, однако, в действительности *не менялась*. И это неудивительно: она представляет ускорение свободного падения, которое, как известно, есть величина *постоянная*.

Для того чтобы явно указать в программе, что величина G не может измениться, перенесем ее описание в отдельный раздел — раздел описания постоянных величин, или *констант* (см. следующий пример). Тем самым мы покажем Паскалю и человеку, который будет читать нашу программу, что эту величину нельзя менять по ходу выполнения программы. Она получает свое значение один раз и не может быть изменена.

Пример 2.11. Программа при использовании констант — более логична и читабельна

```
Program Piza2;

const       { Это раздел описания констант.
              Он находится перед разделом var }
   G=9.8;   { Тип константы определяется автоматически,
              по форме записи числа. В данном случае
              из-за наличия десятичной точки
              это тип real }

var V,H: real;
begin
   write('Введите высоту башни:');
   readln(H); V:=Sqrt(2*G*H);

   writeln('Скорость падения ',V:6:3);
            { Чтобы текст и число не "слиплись",
              после текста внутри апострофов добавлен
              пробел }

   readln
end.
```

Использование констант выполняет еще две функции. Во-первых, описывая величину в разделе констант, мы подстраховываем сами себя, чтобы случайно не изменить ее в программе. (Вам, наверное, это замечание кажется глупым: как можно так ошибиться! Но при написании больших, многостраничных программ это становится актуальным.)

Во-вторых, константы оказываются нужны при объявлении новых типов данных — массивов. Об этом мы поговорим в теме 8.

Задание 2.14. Вычислите длину окружности и площадь круга. Радиус должен вводиться с клавиатуры.

Выводы

1. Данные, с которыми работает программа, хранятся в ячейках. Каждая ячейка имеет имя и тип данных. Изменяемые ячейки называются *переменными*, неизменяемые — *постоянными*.
2. Ячейки, используемые в программе, описываются в разделах `const` и `var`.
3. Для хранения целых чисел используется тип данных `integer`, а для хранения вещественных — `real`.
4. Каждый тип данных имеет свои операции и функции. Особенно это важно для типа `integer`, который имеет две операции деления — `div` и `mod`.
5. Для преобразования значений типа `real` к типу `integer` используются специальные функции — `trunc` и `round`.
6. При записи арифметических выражений нужно помнить о приоритете операций и о типе данных, который получается в результате.
7. Начальные значения переменных можно задавать путем ввода с клавиатуры или с помощью датчика случайных чисел.

Контрольные вопросы

1. Где хранятся все данные, с которыми работает программа?
2. Чем различаются между собой понятия *имя ячейки*, *адрес ячейки* и *значение ячейки*?
3. Какой тип данных используется для хранения целых чисел? А для дробных?
4. Что следует сделать, если в программе используется величина, не изменяющаяся за все время работы программы?
5. В чем различие между операциями `mod`, `div` и `/`?

6. Зачем нужны функции `trunc` и `round`? В чем между ними разница?
7. Какое максимальное значение может принимать переменная типа `integer`? Что делать, если необходимо сохранить целое число, в 10 раз большее этого значения?
8. Как записать на Паскале «2,5 в степени 16,7»?
9. Что означает запись «`1E5`», «`3E-4`», «`.2E7`»?
10. Что нужно использовать, чтобы изменить приоритет выполнения математических операций?
11. Чему равно и какой тип данных имеет выражение `trunc(sqrt(2+52 div 8))-sqr(15 mod 4/3)`?

ТЕМА 3

Учимся работать с символами

В предыдущей теме мы рассмотрели типы данных, позволяющие хранить и обрабатывать числа — целые и дробные. Но, перефразируя известную поговорку, не числами едиными живет программист. Кроме чисел, Паскаль умеет также работать с символьной информацией. Для каждого символа в программе выделяется отдельная ячейка со всеми присущими ячейке параметрами — именем и значением.

Урок 3.1. Как компьютер понимает символы

Под символами мы понимаем буквы и все те значки, которые вы видите на клавиатуре. В Паскале переменные для хранения символов имеют тип `Char`.

За каждым символом закреплен свой числовой код. Все коды сведены в таблицу.

Кодовая таблица ASCII

Обычно для хранения символов используют код, называемый ASCII (американский стандартный код обмена информацией).

Как видите, цифры здесь — не числовые данные, а тоже символы, каждый из которых имеет свой код. В компьютере коды записаны в двоичном виде. На каждый код выделено 8 бит, то есть 1 байт. Получаем $2^8 = 256$ двоичных кодов. Всего в таблице ASCII 256 кодов: наименьшее значение кода 00000000, наибольшее — 11111111 (это 255 в двоичном виде).

Таблица 3.1. Фрагмент таблицы ASCII (таблица кодов символов)

Код	Двоичный код	Символ	Код	Двоичный код	Символ
48	00110000	0	65	01000001	A
49	00110001	1	66	01000010	B
50	00110010	2	67	01000011	C
57	00111001	9	89	01011001	Y
			90	01011010	Z

Описание типа Char и стандартные функции

Пример 3.1. Демонстрация стандартных функций для работы с типом Char

```
Program Letter1;
var
   N: Integer;
   X: Char;
begin
   X:='L';    { В символьную переменную X
                записали символ 'L' }

   writeln(X);

   N:=Ord (X); { Функция Ord возвращает код символа,
                 занесенного в переменную X, то есть код
                 буквы 'L' }

   writeln(N);
   X:='A';
   writeln(X);

   X:=Chr (N); { Функция Chr возвращает символ
                по заданному коду. Сейчас в переменной X
                оказался символ 'L' — именно его код мы
                только что записали в переменную N }

   writeln(X);
   readln
end.
```

При выполнении программа выведет на экран следующее:

```
L
76
A
L
```

Пример 3.2. Ввод символов с клавиатуры

```
Program Letter2;
var
   X,Y:Char;
begin
   writeln('Введите несколько символов:');
   readln(X);
   writeln(X);
   writeln('Введите еще несколько символов:');
   readln(X,Y);
   writeln(X,Y);
   readln
end.
```

Запустив программу на выполнение, введите с клавиатуры последовательность символов (например, ABC) и нажмите Enter. Программа выведет единственный символ:

```
A
```

В ответ на второе предложение введите с клавиатуры CAT. На экране получим следующее:

```
CA
```

ЗАМЕЧАНИЕ

Переменная типа Char принимает только один символ из введенной строки. При вводе символы не заключаются в апострофы. Таким образом, в первом случае из введенных символов запомнился только один, во втором — два.

Можно определять и символьные константы:

```
const Leto='X';
```

Урок 3.2. Тип Char — порядковый тип!

В таблице кодов вы могли заметить такую закономерность:
`'0'<'1'< '2'< '3'< ...< '9'<... 'A'< 'B'< 'C'< ...< 'X' < 'Y' < 'Z'...` (коды символов упорядочены).

Таким образом, для каждого элемента типа `Char` всегда есть предшествующий и последующий элементы. Такой тип данных называется *порядковым*. Тип `Char` — порядковый тип. Тип `Integer` также является порядковым.

Пример 3.3. Стандартные функции, применяемые к порядковому типу

```
Program Letter3;
var
   X1,X2,X3,X4: Char;
begin
   X1:='L';
   writeln(X1);

   X2:=Pred (X1); { Функция Pred возвращает
                    предшествующий элемент
                    относительно значения
                    переменной X1 }

   writeln('Pred=',X2);

   X3:=Succ (X1); { Функция Succ возвращает
                    последующий элемент
                    относительно значения
                    переменной X1 }

   writeln('Succ=',X3);
   readln
end.
```

При выполнении программа выведет на экран следующее:

```
L
Pred=K
Succ=M
```

Задание 3.1. Напишите программу расшифровки 4-буквенного однословного сообщения. Для получения 4 букв нужно ввести 3 строки:

- из первой строки прочесть только первую букву;
- из второй строки прочесть только первую букву;
- из третьей строки прочесть первую и вторую буквы.

Далее расшифровать полученные четыре буквы по такому алгоритму: вместо первой и третьей букв подставить соответственно буквы, отстоящие от них по алфавиту на две буквы назад, а вторую и четвертую буквы оставить без изменения.

Для проверки возьмите пример, приведенный ниже.

Ввод:

```
FINISHED
OR
PENDING?
```

На выводе должно быть слово «DONE».

Задание 3.2. Известно, что коды прописных (заглавных) букв латинского алфавита следуют в таблице непрерывно друг за другом. Коды строчных букв латиницы также следуют непрерывно друг за другом на расстоянии 32 символов от прописных (ниже по таблице). Если `ord('A') = 65`, то `ord('A')+32 = 97`, и это код строчной буквы «a», то есть `chr(ord('A')+32) = 'a'`. Напишите программу, в которой вы вводите прописную букву (только латиницу!), а получаете ее строчный эквивалент, и наоборот, по строчной букве получаете соответствующую прописную.

ЗАМЕЧАНИЕ

С русскими символами такого порядка нет из-за особенностей организации кодовой таблицы. В частности, строчные буквы в таблице следуют не подряд, а с разрывом в середине алфавита.

Выводы

1. Все символы хранятся в компьютере в виде кодов.
2. Обычно для кодирования символов применяется таблица ASCII.
3. Каждому символу соответствует свой код.
4. Для преобразования символов в коды и обратно применяют функции `ord` и `chr`.
5. Тип Char является порядковым типом.

6. Коды буквы латинского алфавита идут последовательно.
7. Русские буквы хранятся в таблице символов ASCII с разрывом в последовательности кодов.
8. Для получения следующего и предыдущего символа используют соответственно функции succ и pred.

Контрольные вопросы

1. Сколько всего различных символов кодируется таблицей ASCII?
2. Какой объем памяти требуется для кодирования одного символа? А для 15 символов?
3. Какой тип данных в Паскале предназначен для хранения символьной информации? Сколько символов можно поместить в одну переменную этого типа?
4. Какой код у буквы «F»? Какой символ кодируется кодом 87?
5. В программе определены 3 переменные (`a,b,c:char;`). В ответ на инструкцию `readln(b,a,c);` пользователь ввел текст `Леша`. В каком месте памяти оказалась каждая из введенных букв?
6. Каков будет результат выполнения инструкции `c:=succ(pred(succ('D')))`?
7. Какое значение получит переменная i в операторе `i := pred(ord('F')-2)`?

ТЕМА 4
Джордж Буль и его логика

В этой теме мы обсудим вопросы, связанные с Решениями. Именно так, с большой буквы. Мы разберем, как же компьютер принимает решения. Конечно, компьютеру не приходится принимать такие сложные решения, как человеку. Однако логики и определенности в поведении компьютера куда больше. Собственно, никаких колебаний у него и не бывает. Каждый раз, когда компьютер принимает решение, оно четко и окончательно — или да, или нет! Согласитесь, людям подобной решимости зачастую не хватает.

Урок 4.1. Необходим еще один тип — логический!

Поговорим о философии, а именно — о логике.

Логика оперирует утверждениями. Любое логическое утверждение может быть либо истинным, либо ложным. При решении многих задач возникает ситуация, когда требуется проверить некоторое условие (сформулированное в виде утверждения) и в зависимости от результата проверки (истинности утверждения) произвести те или иные действия:

- если условие выполняется, то результатом будет «истина»;
- если условие не выполняется, то результатом будет «ложь».

Например, утверждение «4 > 3» является истинным, а утверждение «2 > 3» — ложным.

Такие выражения называются *булевы* (по имени английского математика Джорджа Буля). Область математики, которая изучает действия с булевыми выражениями, называется *булевой алгеброй* или *алгеброй логики*.

Логический тип данных (Boolean)

Для хранения результата проверки условия введен логический тип — `Boolean`. Переменные такого типа называются *булевыми переменными*.

Пример 4.1. Булевы переменные в программе

```
Program Bool1;
var X: integer;
   Bol:Boolean;
begin
   X:=4;
   Bol:=X > 3;   { Это утверждение истинно }
   writeln(Bol);

   Bol:=X < 3;   { Это утверждение ложно }
   writeln(Bol);

   readln
end.
```

При выполнении программы на экране мы получим следующее:

```
TRUE
FALSE
```

Что такое `TRUE` и `FALSE`? Это значения логического выражения: `TRUE` означает «истина», а `FALSE` — «ложь».

Операции отношения

В логических условиях используются операции отношения. Вот как они записываются на языке Паскаль (табл. 4.1).

Таблица 4.1. Запись операций отношения на языке Паскаль

Операция отношения	Запись на языке Turbo Pascal
Меньше	<
Меньше или равно	<=
Больше	>
Больше или равно	>=
Равно	=
Не равно	<>

Ввод-вывод булевских переменных

Булевские переменные можно выводить на экран, но нельзя вводить с клавиатуры. Для этого приходится вводить переменную другого типа, сравнивать ее с образцом и по результатам сравнения устанавливать значение логической переменной.

Пример 4.2. Как ввести с клавиатуры переменную булевского типа

```
program BooleanInput;
var eat:boolean;
   ch:char;
begin
   write('Ты хочешь есть [y/n]?:');
   readln(ch);
   eat:= ch = 'y';
   writeln('Твой ответ:',eat);
   readln
end.
```

Урок 4.2. Логические (булевы) операции

Часто принимаемое решение зависит от результата не одного, а нескольких утверждений. Например, «Вася получит сегодня пятерку, если придет на урок **И** правильно выполнит задание». Значит, нужно научиться объединять результаты нескольких утверждений и принимать общее решение. В приведенном примере этим объединением служит союз «И».

Логическое умножение (конъюнкция)

В алгебре логики операции сравнения в логических выражениях можно комбинировать с помощью логических операций. Обсудим их по порядку.

Рассмотрим утверждение:

$x < 7$ и $x > 3$

Два отношения связаны союзом «и» (and).

Согласно правилам булевой алгебры, комбинация двух логических выражений, связанных между собой союзом «и», всегда является истинной, если истинны оба выражения.

Эта операция называется *логическим умножением*, или *конъюнкцией*. На Паскале она обозначается как and.

Таблица 4.2. Таблица истинности для операции логического умножения

Операнд 1	**Операция**	**Операнд 2**	**Результат**
True	and	True	True
True	and	False	False
False	and	True	False
False	and	False	False

Логическое сложение (дизъюнкция)

Рассмотрим утверждение:

$x > 100$ или $x < 10$

Выражения можно связывать союзом «или» (or).

Логическое выражение, связанное союзом «или», всегда ложно (false), если ложны обе его части. Во всех других случаях результатом будет «истина» (true).

Эта операция называется *логическим сложением*, или *дизъюнкцией*. На Паскале она обозначается как or.

Таблица 4.3. Таблица истинности для операции логического сложения

Операнд 1	**Операция**	**Операнд 2**	**Результат**
True	Or	True	True
True	Or	False	True
False	Or	True	True
False	Or	False	False

Исключающее ИЛИ (сложение по модулю 2)

Рассмотрим утверждение:

либо $x > 5$, либо $x < 0$

Выражения связаны парой «либо-либо».

Такое утверждение истинно, когда истинно только одно из двух составляющих его утверждений.

Можно сформулировать это иначе: логическое выражение истинно (true), если его операнды различны.

Данная операция называется *исключающим ИЛИ*. На Паскале она обозначается как xor.

Таблица 4.4. Таблица истинности для операции исключающего ИЛИ

Операнд 1	**Операция**	**Операнд 2**	**Результат**
True	Xor	True	False
True	Xor	False	True
False	Xor	True	True
False	Xor	False	False

Как еще можно получить эти результаты? Представим значение true как 1 (логическую единицу), а false — как 0 (логический ноль).

Теперь сложим эти значения и возьмем остаток от деления полученного результата нацело на 2 (mod 2). Эта операция называется также *сложением по модулю 2*. Ясно, что результат будет всегда меньше двух.

Логическое отрицание (инверсия)

Рассмотрим утверждение:

не ($x > 100$)

Выражение отрицается частицей «не».

Результат операции противоположен отрицаемому утверждению. Если утверждение было истинным (true), результатом будет «ложь» (false). И наоборот, если утверждение было ложным (false), то получится «истина» (true).

Эта операция называется *логическим отрицанием*, *логическим НЕ* или *инверсией*. На Паскале она обозначается как not.

Таблица 4.5. Таблица истинности для операции логического НЕ

Операнд	**Результат операции not**
True	False
False	True

Применение логических операций в программе

Пример 4.3. Логические операции в программе

```
Program Bool_1;
var X: Integer;
   Bol, OnBol, Rez: Boolean;
Begin
   X:=4;
   Bol:=X>3;
```

```
   OnBol:=X<3;
   writeln('Bol=',Bol);
   writeln('OnBol=',OnBol);
   Rez:=Bol and OnBol;
   writeln('Bol and OnBol=',Rez);
   Rez:=Bol or OnBol;
   writeln('Bol or OnBol=',Rez);
   Rez:=not Bol;
   writeln('not Bol=',Rez);
   readln
end.
```

При выполнении программы имеем на экране следующее:

```
Bol=TRUE
OnBol=FALSE
Bol and OnBol=FALSE
Bol or OnBol=TRUE
not Bol=FALSE
```

Пример 4.4. Составление логических выражений

```
Program Bol_2;
{ Введем логические переменные, которые будут определять
  характеристики студента.
  Составим выражения, определяющие, является ли студент
  первокурсником и получающим стипендию }

var
   Price:Boolean;     { Определяет наличие
                        стипендии у студента }

   Kurs1:Boolean;     { Определяет, является ли
                        студент первокурсником }

   Rezult:Boolean;    { Определяет результат }

begin
   Price:=True;       { Пусть наш студент получает
                        стипендию }

   Kurs1:=True;       { Пусть студент - первокурсник }

   Rezult:=Price and Kurs1;
   writeln('Студент - первокурсник со стипендией? -',
            Rezult);
```

```
  Price:=False;      { Пусть наш студент не получает
                       стипендию }
  Rezult:=Price and Kurs1;
  writeln('Студент - первокурсник со стипендией? - ',
          Rezult);
  readln
end.
```

При выполнении программы имеем на экране следующее:

```
Студент - первокурсник со стипендией? - TRUE
Студент - первокурсник со стипендией? - FALSE
```

Задание 4.1. Определите в программе 4 логических переменных, которые содержат следующую информацию о людях:

`Married` — «истина», если человек женат (замужем);

`Blond` — «истина», если у человека светлые волосы;

`Male` — «истина», если человек — мужчина;

`Employed` — «истина», если человек работает.

Составьте логические выражения, с помощью которых можно определить, является ли человек:

1) замужней женщиной;
2) неженатым мужчиной;
3) незамужней блондинкой;
4) безработной незамужней женщиной;
5) либо неженатым, либо безработным, либо и тем и другим.

Приоритет логических операций

При объединении нескольких логических операций нужно помнить, что они, подобно математическим операциям, подчинены правилам приоритета (табл. 4.6). Для изменения приоритета выполняемых действий необходимо использовать скобки.

Таблица 4.6. Приоритет логических операций

Приоритет	**Логическая операция**
1 (самый высокий)	`not`
2	`and`
3 (самый низкий)	`or, xor`

Операции xor и or имеют одинаковый приоритет, и значит, при отсутствии скобок, выполняются слева направо.

ЗАПОМНИТЕ!

Это очень важно! Приоритет любой операции сравнения ниже, чем у любой логической операции. Это значит, что при объединении сравнений при помощи логических операций каждое сравнение необходимо взять в скобки. Например, утверждение `2 ≤ x ≤ 4` *должно быть записано как* `(2<=x)` *and* `(x<=4)`.

Выводы

1. Существуют утверждения, относительно которых можно однозначно сказать, истинны они или ложны.
2. Для записи результатов таких утверждений, а также для анализа условий необходим еще один тип данных — `Boolean`.
3. Данные этого типа могут принимать одно из двух значений: true («истина») или `false` («ложь»).
4. При составлении выражения для анализа условий используются операции отношений: >, <, >=, <=, =, <>.
5. Для объединения нескольких логических утверждений в одно используются логические операции (`and, or, xor, not`), результаты которых формируются в соответствии с таблицами истинности.
6. Порядок выполнения действий определяется скобками и приоритетами операций.

Контрольные вопросы

1. Что такое логическое утверждение?
2. Сколько различных значений могут принимать логические утверждения? Как они обозначаются?
3. Как записать на Паскале утверждение «икс не равен пяти»?
4. Как записать на Паскале утверждение «игрек не принадлежит отрезку [3,5]»?
5. Чему равен результат операции `true or false and false`?
6. Правильно ли записано выражение `(x<0) or (x+2)>3`?
7. Чему будет равен результат операции `true xor false xor true`?

ТЕМА 5

Анализ ситуации и последовательность выполнения команд

Все алгоритмы и программы, которые мы до настоящего момента рассматривали, были *линейными*, то есть выполнялись последовательно, шаг за шагом, инструкция за инструкцией, независимо от введенных данных. Однако часто бывает необходимо выполнять разные действия в зависимости от того, какое решение было принято. В данной теме мы рассмотрим, как менять порядок выполнения команд по результатам проверки некоторого условия.

Урок 5.1. Проверка условия и ветвление в алгоритме

В предыдущей теме мы познакомились с логическими выражениями. Именно они используются в Паскале для организации ветвления.

Пусть стоит такая задача:

Если `X > 3`, то выводим на экран `X`.

Блок-схема алгоритма решения этой задачи выглядит так (рис. 5.1):

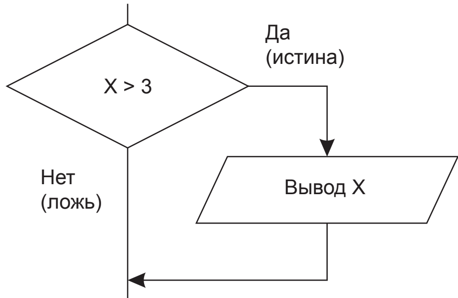


Рис. 5.1. Блок-схема алгоритма, выводящего число, если оно больше трех

На языке Паскаль такую схему обрабатывает условный оператор if.

Полная и неполная форма оператора if

Формат записи оператора if следующий:

```
if <условие> then <оператор>
```

Пример:

```
if X > 3 then writeln(X);
```

Под условием здесь понимается любое выражение, результат которого имеет тип boolean.

Это *неполная форма алгоритма с ветвлением*.

Изменим немного нашу задачу:

Если `X >= 3`, то вывести на экран `X`, иначе вывести текст `'X < 3'`.

В том и другом случае `X` необходимо увеличить на 1.

Здесь используется расширенный условный оператор — *полная форма ветвления*: `if ... then ... else ....`

Формат записи оператора: `if <условие> then <оператор-да> else <оператор-нет>`. (См. блок-схему на рис. 5.2).

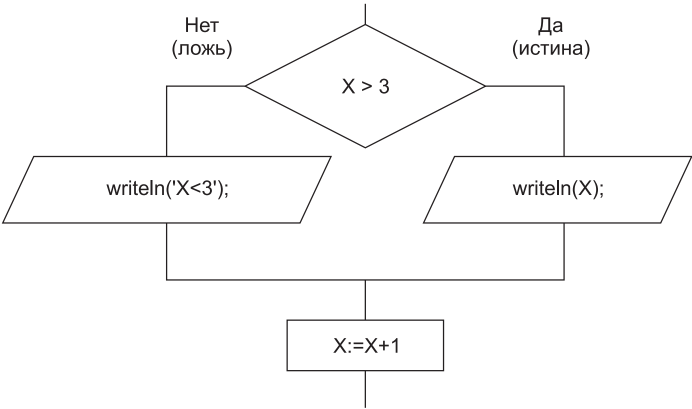


Рис. 5.2. Блок-схема алгоритма, выводящего число, если оно больше трех, или сообщение «X<3» в противном случае

В общем случае структурная схема условного оператора выглядит так (рис. 5.3).

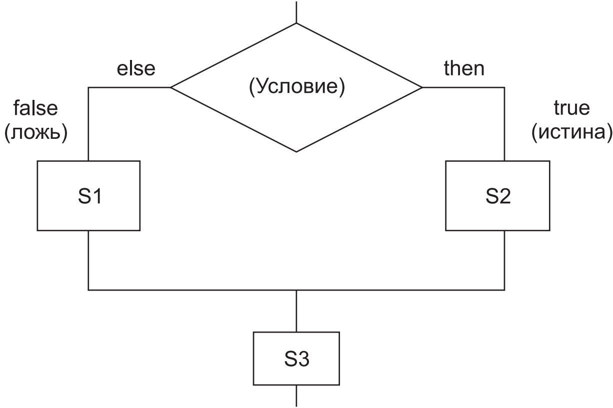


Рис. 5.3. Структурная схема условного оператора If

(S1, S2, S3 — условные обозначения операторов.)

ЗАПОМНИТЕ!

Перед `else` *нельзя ставить точку с запятой!*

Независимо от формы записи условия, после окончания оператора `if` программа снова «соединяется» и продолжает выполнять операторы, стоящие после структуры `if`. Это наглядно демонстрирует рис. 5.3. Оператор `S3` выполняется независимо от того, каким будет результат проверки условия.

Рассмотрим задачу определения количества корней квадратного уравнения по дискриминанту (рис. 5.4).

Пример 5.1. Анализ дискриминанта квадратного уравнения

```
program Diskr;
var
   A,B,C,D: real;
begin
   write('Введите коэффициенты A,B,C:');
   readln(A,B,C);
   D:=SQR(B)-4*A*C;
   if D >= 0 then
      if D > 0 then
         writeln('Два вещественных корня')
```

```
      else
         writeln('Один вещественный корень')
   else
      writeln('Нет вещественных корней');
   readln
end.
```

В этой задаче используется вложенный оператор `if`.

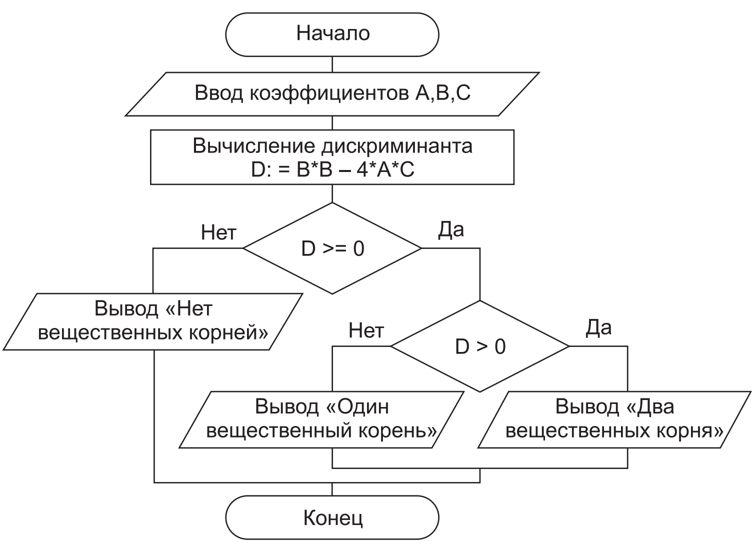


Рис. 5.4. Блок-схема алгоритма, определяющего количество корней квадратного уравнения по дискриминанту

ЗАПОМНИТЕ!

При использовании вложенных операторов `if` *слово* `else` *относится к последнему* `if`, *у которого еще нет* `else`.

Оформление программ

Возможно, вы уже обратили внимание, что во всех приводимых нами примерах мы слегка (на 2–3 позиции) сдвигаем операторы вправо. Таким способом мы выделяем, например, блок описания переменных var, а также операторы основной программы относи-

тельно begin и end. Это мы делаем сознательно и хотели бы рекомендовать вам поступать так же. Паскаль прекрасно поймет вашу программу, даже если вы ее всю наберете в одну строку. Однако через несколько дней в ней будет тяжело разобраться даже автору, не говоря уже о других людях.

Мы рекомендуем всегда сдвигать вправо вложенные фрагменты программы относительно точки вложения. Так, например, список переменных, определяемых в разделе var, следует сдвигать относительно слова var, список операторов основной программы — относительно begin и end, а операторы, вложенные в структуру if, — относительно if и else.

При этом соответствующие пары операторов begin и end рекомендуется располагать друг под другом, на одинаковом расстоянии от левого края. Это позволяет в сложной программе отследить, какому оператору begin какой оператор end соответствует, и, например, найти пропущенный оператор.

ЗАМЕЧАНИЕ

Рекомендуется при наборе программы сразу же после оператора begin *писать оператор* end, *и потом уже между ними вставлять вложенные операторы. Это позволяет избежать ситуаций с появлением непарных* begin/end.

Этим методом рекомендуется также пользоваться при наборе апострофов и скобок: набрав левую скобку, сразу поставьте правую и после вписывайте текст между ними.

Задание 5.1. Нарисуйте блок-схему алгоритма и напишите программу, которая анализирует введенное с клавиатуры число и выдает на экран:

- удвоенное значение числа, если число положительное;
- абсолютное значение числа, если число отрицательное.

Задание 5.2. Нарисуйте блок-схему алгоритма и напишите программу, которая анализирует введенное с клавиатуры число на четность и сообщает о результате. Используйте операцию нахождения остатка от деления числа на 2.

Урок 5.2. Блоки операторов

Управляющая структура `if ... then` может показаться негибкой, так как после служебного слова `then` должен стоять только один оператор. Если вы напишете два оператора подряд (например, `if x<>0 then y:=1/x; x:=0;`), то второй оператор выполнится в любом случае, независимо от проверяемого условия.

Если требуется выполнить последовательность действий (несколько операторов подряд), то ее заключают в блок, образуемый операторами `begin` и `end`.

Пример:

```
if X > 3 then begin S1; S2; S3; S4 end;
```

Здесь `S1-S4` символически обозначают операторы.

Эта группа (`begin S1; S2; S3; S4 end`) называется *составным оператором*, или *операторной скобкой*. Она как бы говорит компилятору, что данный блок операторов нужно рассматривать как единое целое.

Пример 5.2. Вычисление корней квадратного уравнения

```
Program Quad;
var
   A,B,C: real;{ Переменные для хранения коэффициентов }
   D: real;    { Переменная для дискриминанта }
   X1,X2: real;{ Переменные для получения корней }
begin
   writeln('Введите коэффициенты A, B, C:');
   readln(A,B,C);
   D:=Sqr(B) - 4*A*C;
   if D < 0 then
      writeln('Уравнение не имеет вещественных корней')
   else
      if D=0 then
         writeln('У уравнения один корень',
                  -B/(2*A):6:2)
      else

      { Ниже идет составной оператор }
      begin
         X1:=(-B + Sqrt(D))/(2*A);
         X2:=(-B - Sqrt(D))/(2*A);
         writeln('У уравнения два корня:',
```

```
                    X1:6:2, X2:6:2)
        end;

    readln
end.
```

Задание 5.3. Нарисовать блок-схему алгоритма и написать программу, в которой с клавиатуры вводится код режима работы Kd и цвет Cl. На экран выводится сообщение. При этом, если Kd = 0, то сообщение выводится символами цвета Cl, а если Kd = 1, то сообщение выводится символами цвета Cl+16.

На экране во втором случае вы получите мигающие символы, так как код их цвета больше 15. В этом случае в качестве номера мигающего цвета Turbo Pascal использует остаток от деления кода цвета нацело на 16.

ЗАМЕЧАНИЕ

В качестве проверяемых условий можно использовать логические операции.

Рассмотрим задачу: ввести три числа A, B, C и определить, равны ли введенные числа. Блок-схема алгоритма показана на рис. 5.5.

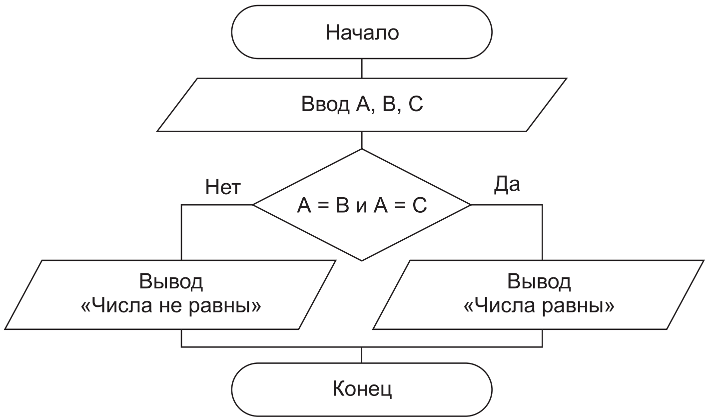


Рис. 5.5. Блок-схема алгоритма, определяющего, равны ли друг другу три введенные с клавиатуры числа

Пример 5.3. Проверка равенства трех чисел, введенных с клавиатуры

```
Program Mx;
var
   A,B,C: integer;
begin
   writeln(' Введите 3 числа:');
   readln(A,B,C);
   if (A = B) and (A = C) then
      writeln('Числа равны')
   else
      writeln('Числа не равны');
   readln
end.
```

Задание 5.4. Ввести три числа A, B, C и определить максимальное из них. Нарисуйте блок-схему алгоритма и напишите программу.

Задание 5.5. Ввести четырехзначное целое число и определить, является ли оно палиндромом, или «перевертышем» (такими, например, являются числа 6666 и 3223). Нарисуйте блок-схему алгоритма.

Подсказка: Для выделения отдельных разрядов числа используются операции div и mod.

Задание 5.6. Ввести три числа A, B, C. Если ни одно из чисел не равно нулю, то в переменную K записать среднее арифметическое трех чисел.

Задание 5.7. Введите значение X и, используя график функции (рис. 5.6), определите значение Y. Заполните блок-схему алгоритма (рис. 5.7).

Задание 5.8. Положение фигуры на шахматной доске (8×8) описывается двумя числами — номером горизонтали и номером вертикали. Ввести с клавиатуры координаты ферзя (X, Y) и координаты любой фигуры (M, N). Проверить, находится ли фигура под ударом. Ферзь бьет по вертикали, горизонтали и диагонали.

Задание 5.9. Введите число с клавиатуры. Если это число четное и кратно 7, то выведите свое имя на экран красным (red) цветом, иначе выведите его зеленым (green) цветом. Предварительно заполните блок-схему алгоритма (рис. 5.8).

Задание 5.10. Ввести три числа A, B, C и определить среднее из них (то, которое больше одного, но меньше другого). Нарисуйте блок-схему алгоритма и напишите программу.

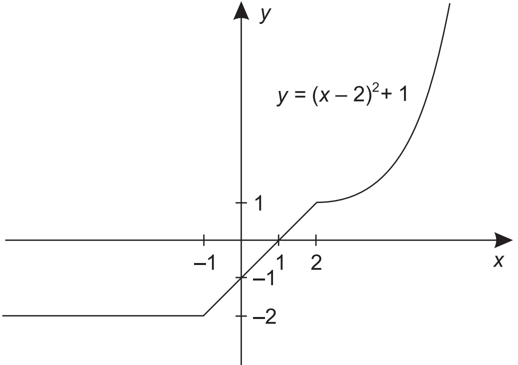


Рис. 5.6. График кусочно-заданной функции для задания 5.7

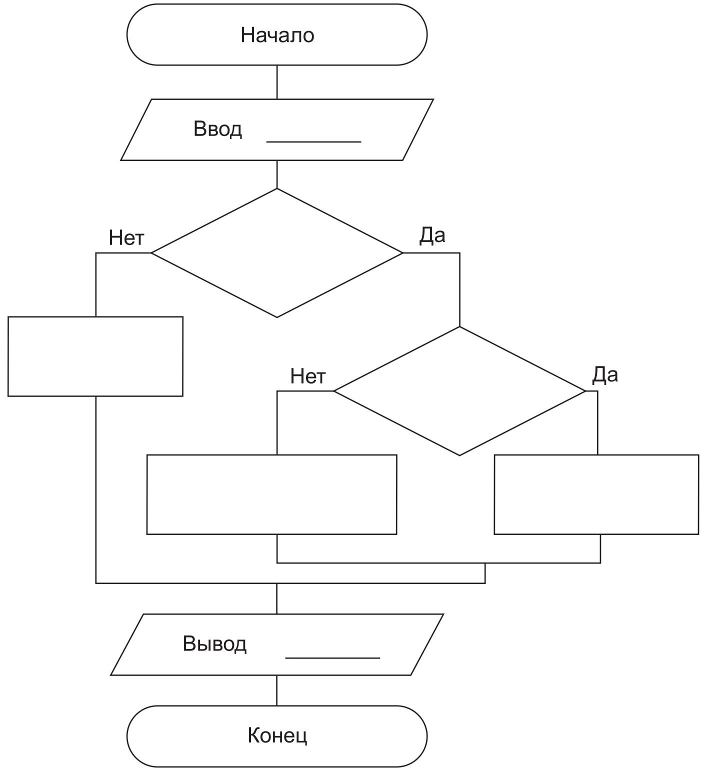


Рис. 5.7. «Слепая» блок-схема алгоритма вычисления кусочно-заданной функции, изображенной на рис. 5.6

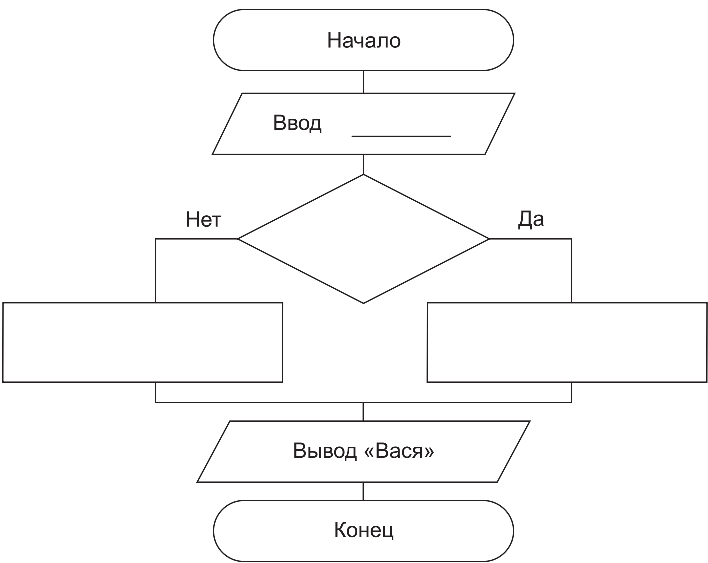


Рис. 5.8. «Слепая» блок-схема алгоритма, выводящего слово «Вася» красным цветом на экран, если введенное число четное и кратно 7, и зеленым цветом в противном случае

Урок 5.3. Ветвление по ряду условий (оператор case)

Оператор `if` позволяет выполнять переходы на ту или иную ветвь по значению булева (логического) условия. Используя несколько операторов `if`, можно производить ветвление по последовательности условий.

Пример 5.4. Преобразование введенного целого числа из диапазона (0..4) в его словесное представление

```
Program Digit1;
var
   Num: integer;
begin
   write('Введите число:');
   readln(Num);
   if Num = 0 then
      writeln('Ноль');
```

```
   if Num = 1 then
      writeln('Один');
   if Num = 2 then
      writeln('Два ');
   if Num = 3 then
      writeln('Три ');
   if Num = 4 then
      writeln('Четыре');
   readln
end.
```

Выполним ту же задачу, используя другую управляющую структуру — оператор выбора `case ... of`.

Формат записи оператора таков:

```
Case <выражение порядкового типа> of
<значение1>:<оператор1>;
…
<значениеN>:<операторN>
else <оператор>
end
```

Пример 5.5. Использование структуры case ... of для перевода целого числа в его словесное представление

```
Program Digit2;
var Num: integer;
begin
   write('Введите число:');
   readln(Num);
   case Num of
     0: writeln('Нуль');
     1: writeln('Один');
     2: writeln('Два ');
     3: writeln('Три ');
     4: writeln('Четыре')
     else writeln('Введено другое число')
 end;
 readln
end.
```

Далее приведена блок-схема алгоритма решения этой задачи (рис. 5.9).

Переменная `Num` является *селектором* в операторе `case`. По значению селектора происходит переход на соответствующую метку.

Селектор должен принадлежать к порядковому типу (то есть он не может иметь тип `real`)!

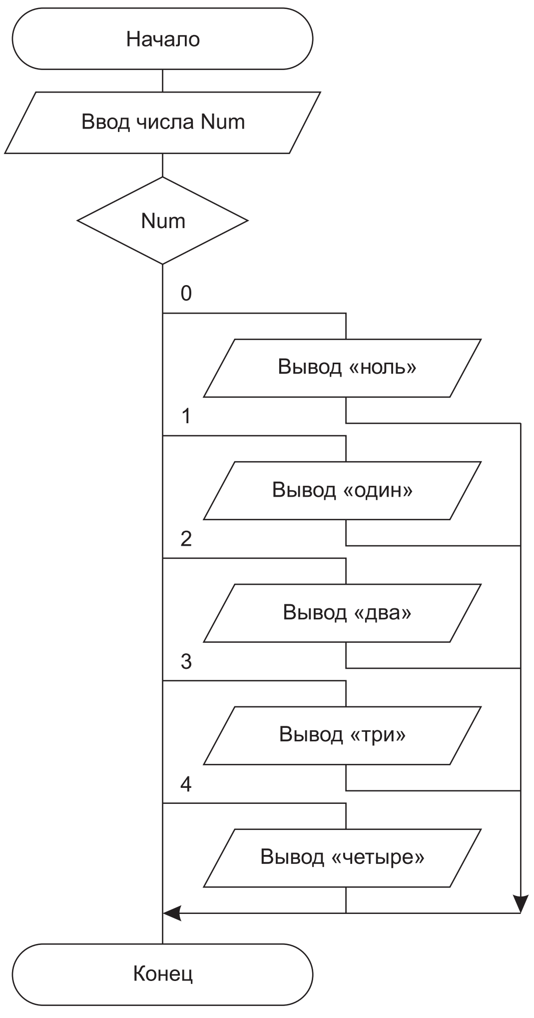


Рис. 5.9. Блок-схема алгоритма, выводящая словесное представление введенного с клавиатуры числа (не большего 4)

Пример 5.6. Определение номера квартала по введенному номеру месяца

```
Program Digit3;
var
   Num:integer;
begin
   write('Введите номер месяца:');
   readln(Num);
   case Num of
      1,2,3 : writeln('Первый квартал');
      4,5,6 : writeln('Второй квартал');
      7..9  : writeln('Третий квартал'); { метка 7..9 —
                                           интервал чисел
                                           от 7 до 9;
                                           тоже самое,
                                           что и 7,8,9 }
      10..12: writeln('Четвертый квартал')
      else writeln('Некорректный ввод')
   end;
   readln
end.
```

При переходе на метку может выполняться целый блок операторов, который оформляется с помощью структуры

```
begin ... end.
```

Задание 5.11. Написать программу, в которой в переменную типа Char вводится символ с клавиатуры. Программа выдает сообщение о том, какой символ был введен:

- цифра от 0 до 9;
- латинская строчная буква;
- латинская срочная буква.

При записи меток в операторе case можно использовать интервальный тип. Например, интервал для латинских заглавных символов записывается: 'A'..'Z'.

Если нужно учесть строчную латиницу, то интервал для меток будет выглядеть так: 'A'..'Z','a'..'z'.

Выводы

1. Существуют задачи, решение которых включает анализ логического условия. Такие задачи описываются разветвляющимся алгоритмом (сравните с линейным алгоритмом).
2. При ветвлении анализируется логическое выражение и, в зависимости от его результата, выполняется та или иная ветвь алгоритма.
3. На Паскале оператор ветвления называется `if`. Он имеет две формы записи — полную и неполную.
4. При полной форме записи `if` в случае истинности логического условия выполняется один блок программы (после слова `then`), а в случае ложности — другой (после слова `else`).
5. При неполной записи оператора `if` блок `else` опускается.
6. При переходе на ту или иную ветвь алгоритма после анализа логического условия возможно выполнение блока операторов, который оформляется с помощью структуры `begin ... end`.
7. Точка с запятой слева и справа от `then` и от `else` не ставится.
8. В случае, когда анализируемое выражение может иметь более двух значений, и при разных значениях нужно выполнять разные инструкции, используют оператор `case`.
9. Оператор `case` должен заканчиваться ключевым словом `end`. Это один из тех редких случаев, когда количество операторов `begin` в программе не будет совпадать с количеством операторов `end`.
10. Чтобы текст программы был более понятен, вложенные (подчиненные) блоки операторов принято оформлять со сдвигом вправо, лесенкой. При каждом следующем вложении операторы сдвигают еще на несколько позиций вправо.

Контрольные вопросы

1. Чем отличается линейный алгоритм от ветвления?
2. Какие ключевые слова используются в Паскале для организации ветвления? Что находится между ними?
3. Чем полное ветвление отличается от неполного?

4. Как оформлять текст программы, чтобы он был понятнее?
5. Что необходимо использовать, если в случае истинности некоторого условия нужно выполнить несколько операторов?
6. Как быть, если в случае истинности некоторого условия никаких действий выполнять не требуется, а в случае ложности нужно выполнить несколько действий?
7. Какую управляющую структуру Паскаля нужно использовать, если проверяемое выражение может принимать несколько возможных значений, и в каждом случае необходимо выполнить разные действия?
8. В каком случае количество операторов `begin` в программе не должно соответствовать количеству операторов `end`?

ТЕМА 6

Многократно повторяющиеся действия

Иногда необходимо повторить определенные действия в программе. Повторение некоторой последовательности действий называется *циклом*. Саму последовательность повторяющихся действий называют *телом цикла*.

Если число повторений известно заранее, то используется структура, которая называется *циклом с заданным (известным) числом повторений*, или *циклом со счетчиком*. Этот вид цикла является частным случаем цикла с условием. Мы начинаем с этого вида цикла в силу его простоты и наглядности.

Урок 6.1. Оператор цикла for

На языке Паскаль повторение некоторой последовательности действий известное число раз выполняет оператор `for`. Подсчет количества выполняемых действий осуществляется при помощи специальной переменной — *счетчика*. Поэтому цикл `for` называют иногда *циклом со счетчиком*. Цикл `for` на Паскале может быть представлен в двух формах. Первая форма последовательно наращивает счетчик:

```
for <переменная порядкового типа>:=<начальное значение> to
<конечное значение> do <оператор>
```

Вторая форма последовательно уменьшает счетчик:

```
for <переменная порядкового типа>:=<начальное значение> downto
<конечное значение> do <оператор>
```

Оператор for с последовательным увеличением счетчика

Пример 6.1. Вывод на экран квадратов чисел от 1 до 10

```
Program Test1;
var N: integer;
begin
   for N:=1 to 10 do    { Переменная N будет меняться
                          от 1 до 10 с шагом 1 }

      writeln(sqr(N));  { Эта строка — тело цикла.
                          Оно выполняется 10 раз }

 readln
end.
```

Поясним пример 6.1. Переменная N является *счетчиком цикла*. Счетчик цикла всегда должен иметь порядковый тип (то есть он не может иметь тип real). В операторе for указаны его начальное и конечное значения. Начальное значение не обязательно равно 1! При первом выполнении тела цикла N = 1, при втором — N = 2 и т. д. При последнем выполнении тела цикла N = 10. Каждый раз перед выполнением тела цикла текущее значение N сравнивается с конечным. После каждого выполнения тела цикла переменная N увеличивается на 1. Блок-схема алгоритма представлена на рис. 6.1.

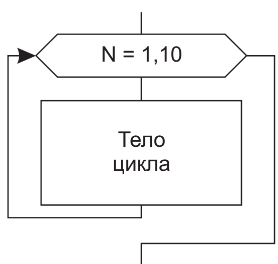


Рис. 6.1. Блок-схема организации цикла в примере 6.1

Как только N превысит конечное значение, выполнение цикла прекращается. Считается, что после окончания цикла переменная цикла не определена (то есть в разных реализациях языка Паскаль она может принимать разные значения). Иными словами, непра-

вильно считать, что после окончания цикла переменная-счетчик цикла имеет какое-то определенное значение.

Крайне не рекомендуется внутри цикла самостоятельно менять счетчик цикла, особенно в сторону уменьшения. Это может привести к «зацикливанию» программы (бесконечному повторению тела цикла).

Оператор for с последовательным уменьшением счетчика

Счетчик может изменяться с шагом –1. Это вторая форма оператора for (for … downto … do).

Пример 6.2. Вывод на экран кубов чисел от 11 до 5

```
Program Test2;
var
   N: integer;
begin
   for N:=11 downto 5 do   { Счетчик N изменяется
                             с шагом -1 }

      write(N*N*N:5);      { Эта строка — тело цикла;
                             оно выполняется 8 раз,
                             так как N изменяется
                             от 11 до 5 с шагом -1 }

   writeln;                { Этот оператор нужен,
                             чтобы закончить вывод чисел
                             в одну строку }
   readln
end.
```

Урок 6.2. Применение циклов со счетчиком

Можно организовать выполнение одного цикла внутри другого. В этом случае различают внешний и внутренний циклы — например, когда при каждом значении счетчика внешнего цикла нужно несколько раз выполнить какое-то действие (внутренний цикл). Счетчик внешнего цикла изменяется медленнее, чем счетчик внутреннего.

Цикл в цикле

Рассмотрим задачу вывода последовательности пар чисел:

1 1
1 2
1 3
1 4
2 1
2 2
2 3
2 4
3 1
3 2
3 3
3 4

Блок-схема алгоритма решения задачи показана на рис. 6.2.

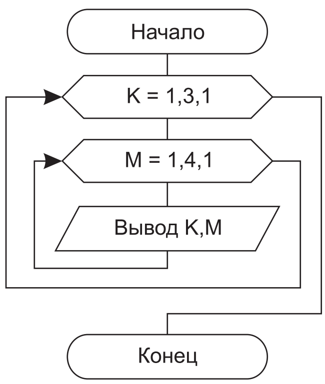


Рис. 6.2. Блок-схема алгоритма с вложенными циклами, выводящего последовательность из примера 6.2

Пример 6.3. Использование цикла в цикле

```
Program Test3
var
   K,M: integer;
begin
   for K:=1 to 3 do
      for M:=1 to 4 do
```

```
        writeln(K,' ',M); { Пробел в апострофах между
                            K и M нужен для того,
                            чтобы эти числа не
                            сливались друг с другом }

  readln
end.
```

Для каждого значения переменной K переменная M меняется от 1 до 4. Нетрудно подсчитать, что в этом случае оператор writeln выполнится 12 раз.

Задание 6.1. Вывести на экран 6 раз свое имя.

Задание 6.2. Вывести на экран таблицу умножения для 5 чисел от 9 до 4.

Задание 6.3. Вывести на экран коды таблицы ASCII от 0 до 255 и их символы. Выводить парами код и символ.

Трассировка

Для понимания чужой программы и для проверки правильности написания своей используют метод пошагового выполнения программы с отслеживанием значений всех переменных.

Пример 6.4. Вычисление суммы чисел от 6 до 10

```
Program Test4;
var
   N: integer;   { Это будет счетчик цикла for }

   S: integer;   { В этой переменной будем
                   накапливать сумму }
begin
   S:=0;         { Первоначально обнулим сумматор }

   for N:=6 to 10 do

      S:=S + N;  { Эта строка – тело цикла.
                   При его выполнении каждый раз
                   к S прибавляется очередное N.
                   Переменную S можно сравнить
                   с аккумулятором, в котором
                   накапливается сумма }
```

```
   writeln('Сумма чисел=', S:6);
   readln
end.
```

Блок-схема алгоритма решения задачи показана на рис. 6.3.

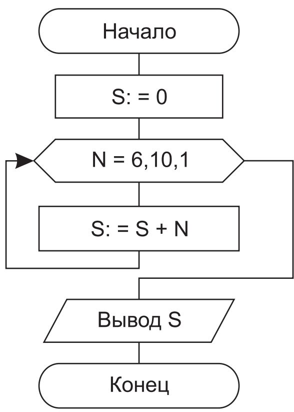


Рис. 6.3. Блок-схема алгоритма вычисления суммы чисел от 6 до 10

Для проверки правильности работы программы рекомендуется пошагово отслеживать изменение всех переменных после выполнения каждого оператора программы. Такой процесс называется *трассировкой*. Продемонстрируем этот прием (табл. 6.1).

Таблица 6.1. Трассировка программы из примера 6.4

Оператор	Условие	N	S	Примечание
S:=0			0	
for N:= 6 to 10 do	Да	6		
S:= S + N			6	0 + 6 = 6
For N:= 6 to 10 do	Да	7		
S:= S + N			13	6 + 7 = 13
For N:= 6 to 10 do	Да	8		
S:= S + N			21	13 + 8 = 21
For N:= 6 to 10 do	Да	9		

Оператор	Условие	N	S	Примечание
S:= S + N			30	21 + 9 = 30
For N:= 6 to 10 do	Да	10		
S:= S + N			40	30 + 10 = 40
For N:= 6 to 10 do	Нет	11		
writeln('Сумма чисел ', S:3)		???		На экране: Сумма чисел = 40

В результате работы программы на экране получим число 40.

Для операторов, выполняющих проверку условий (if, for и т. п.) в столбце «Условие» принято указывать результат проверки. В данном случае в цикле for проверяется условие продолжения цикла.

Символы «???» подчеркивают, что значение счетчика цикла по выходе из цикла считается неопределенным.

Метод трассировки очень помогает при отладке программы, когда программа выдает не тот результат, который должна выдать. Осуществляя пошаговую трассировку, мы вникаем в логику работы программы и на каждом шаге проверяем, правильны ли были наши рассуждения при ее написании.

Вычисление суммы ряда

Рассмотрим задачу вычисления суммы ряда:

$$\frac{1}{1\times1}+\frac{1}{2\times2}+\frac{1}{3\times3}+\frac{1}{4\times4}+\frac{1}{5\times5}.$$

Здесь мы имеем ряд дробей, у которых в знаменателях записаны квадраты чисел от 1 до 5.

Рассмотрим каждую дробь как произведение двух дробей, например:

$$\frac{1}{3\times3}=\frac{1}{3}\times\frac{1}{3}.$$

В общем виде это можно записать так:

$$\frac{1}{N\times N}=\frac{1}{N}\times\frac{1}{N}.$$

Блок-схема алгоритма решения задачи представлена на рис. 6.4.

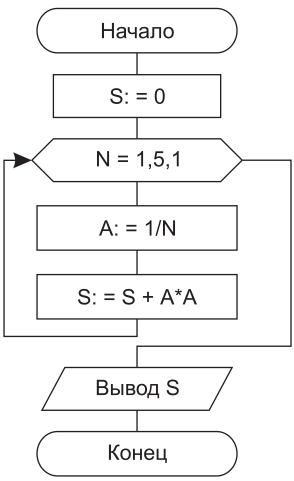


Рис. 6.4. Блок-схема алгоритма вычисления суммы ряда

Пример 6.5. Вычисление суммы ряда

```
Program Test5;
var
   N: integer;      { Это будет счетчик цикла for }

   S: real;         { В этой переменной будем
                      накапливать сумму.
                      Выбрали для нее тип real,
                      так как искомая сумма будет
                      нецелым числом }

   A: real;         { Для записи очередной дроби 1/N
                      (тоже число не целое) }
begin
   S:=0;            { Первоначально обнулим сумматор }

   for N:=1 to 5 do
   begin
      A:=1 / N      { Эти строки — тело цикла.
                      В цикле необходимо выполнить
                      два оператора, поэтому объединяем
                      их в блок  begin ... end }
```

```
    S:=S + A*A;
  end;
  writeln('Сумма чисел=', S:6:4);
  readln
end.
```

Задание 6.4. Написать программу вычисления $n!$ (факториал числа n), где n — целое неотрицательное число.

Определение *факториала*:

$0! = 1$

$1! = 1$

$2! = 1 \cdot 2$

$3! = 1 \cdot 2 \cdot 3$

. . . .

$n! = 1 \cdot 2 \cdot 3 \cdot .... \cdot (n - 1) \cdot n$

Другими словами, $n!$ — это произведение первых n натуральных чисел.

Каждый следующий результат (обозначим его P) получается путем умножения предыдущего результата (предыдущего P) на счетчик, который пробегает значения от 1 до n. Обозначим значение счетчика буквой k. Получаем общий вид выражения: $P = P \cdot k$ (то есть воспользуемся рекуррентной формулой вычисления факториала: $n! = (n - 1)! \cdot n$).

Программа должна быть организована так: с клавиатуры вводится число n (n — положительно), а затем на экран выдается таблица факториалов чисел до n включительно.

Задание 6.5. Написать программу вычисления суммы ряда $S = 1 + 2 + 3 + 4 + 5 + 6$. Нарисовать блок-схему и заполнить таблицу трассировки. Убедиться при трассировке, что сумма равна 21.

Таблица 6.2. Заготовка для таблицы трассировки алгоритма из задания 6.5

Оператор	Условие	S	K (счетчик)	Примечание

Задание 6.6. Написать программу вычисления суммы ряда для n слагаемых (n вводится с клавиатуры):

$$\frac{1}{1\times2\times3}+\frac{1}{2\times3\times4}+\frac{1}{3\times4\times5}+\frac{1}{4\times5\times6}+\cdots$$

Задание 6.7. Используя возможности модуля Crt для работы с экраном в текстовом режиме, написать программу, которая 16 раз меняет цвет экрана и выводит любой текст на новом фоне в центр экрана.

Пояснение: разумно, если цвет фона и параметр цикла будут одной переменной (палитра цветов изменяется в диапазоне 0–15).

Задание 6.8. Используя возможности модуля Crt, напишите программу, в которой символ «звездочка» (*) пробегает по всему периметру экрана из верхнего левого угла.

Пояснение: в программе организуйте 4 цикла. В качестве счетчика используйте координаты `X` и `Y`. Нарисуйте блок-схему алгоритма.

Попробуйте изменить программу, используя всего два цикла: в одном цикле звездочки бегут сразу по верхней и нижней строкам экрана, в другом — сразу по левому и правому краю. Пусть каждая следующая звездочка выводится случайным цветом.

Задание 6.9. По экрану разбросайте 1000 звездочек в случайном месте случайным цветом с небольшой задержкой. Не забудьте инициализировать датчик случайных чисел в начале программы — один раз! Нарисуйте блок-схему алгоритма.

Выводы

1. Для организации многократно повторяющихся действий с заранее известным числом повторений используется оператор цикла `for`.
2. Счетчик цикла всегда имеет порядковый тип.
3. Счетчик цикла изменяется с шагом +1, если оператор имеет форму

```
for ...:=... to .... do
```

4. Счетчик цикла изменяется с шагом −1, если оператор имеет форму

   ```
   for ...:=... downto .... do
   ```

5. Чтобы узнать, сколько раз выполнится тело цикла `for`, нужно найти разность между крайними значениями счетчика (по модулю) и прибавить к результату 1.
6. Не рекомендуется изменять счетчик цикла в теле цикла.
7. Если внутри цикла `for` поставить еще один цикл `for`, то количество раз, которое выполнится тело внутреннего цикла, равно произведению числа повторений внешнего цикла на число повторений внутреннего.
8. Для проверки правильности работы алгоритма его выполняют вручную, шаг за шагом, отслеживая изменения всех переменных. Это называется трассировкой.

Контрольные вопросы

1. Какой оператор нужно использовать, чтобы вывести в каждой строке экрана слово «Привет»?
2. Чем отличаются формы `to` и `downto` оператора `for`?
3. Переменные какого типа должны использоваться в качестве счетчика цикла `for`?
4. Сколько раз выполнится тело внутреннего цикла:

   ```
   for i:=2 to 6 do
      for j:=5 downto 3 do
         writeln('*');
   ```

5. Предположим, некоторая написанная программа выдает странный результат. Вероятно, программа написана с ошибкой. Как понять, где содержится ошибка?
6. Требуется последовательно присвоить переменной `N` значения всех трехзначных чисел. Напишите оператор, присваивающий переменной `N` нужные значения.

ТЕМА 7
Циклы с условием

Цикл со счетчиком for, рассмотренный в предыдущей теме, отлично выполняет свои функции, когда число повторений тела цикла известно к моменту его начала (или известны начальное и конечное значения счетчика, что, впрочем, то же самое). Однако такая «радужная» картина встречается в программировании далеко не всегда. Часто приходится решать задачи, когда число повторений цикла неизвестно и определяется лишь постепенно, после некоторого количества повторений тела цикла. В этом случае применяют другую разновидность цикла — *цикл с условием*. В языке Паскаль циклов с условием предусмотрено два: условие цикла может проверяться перед телом цикла или после него.

Урок 7.1. Цикл с предусловием

В первой разновидности цикла условие проверяется перед выполнением тела цикла. Поэтому данное условие правильно будет назвать *условием продолжения цикла*. Цикл такого вида называется *циклом с предусловием*.

Цикл будет повторяться до тех пор, пока проверка этого условия будет давать результат «истина» (true), то есть пока условие выполняется. Если условие сразу оказывается ложным, цикл не будет выполнен ни разу.

Описание цикла с предусловием

Запишем цикл с предусловием на языке блок-схем (рис. 7.1).

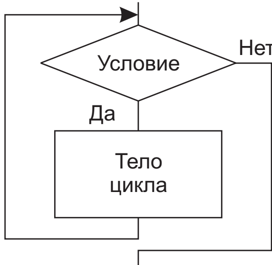


Рис. 7.1. Блок-схема цикла с предусловием

Вот как этот тип цикла реализуется на языке Паскаль:

```
While <логическое условие> do <оператор-тело_цикла>
```

Так же как при использовании цикла `for` и оператора `if` после служебного слова `do` предполагается только один оператор.

Если в теле цикла нужно выполнить несколько операторов, оно оформляется как блок `begin … end`.

ЗАПОМНИТЕ!

После служебных слов `then, else, do` *(в операторах* `if, for, while`*) должен стоять только один оператор! Если необходимо выполнить несколько операторов, они должны быть взяты в операторные скобки (перед операторами нужно поставить* `begin`*, после —* `end`*).*

Точка с запятой не ставится ни перед служебными словами `then, else, do`*, ни после них!*

Приближенное вычисление суммы бесконечного ряда

Рассмотрим задачу, в которой требуется написать программу приближенного вычисления суммы:

$$1+\frac{x}{1!}+\frac{x^2}{2!}+\frac{x^3}{3!}+$$

с точностью ε.

Считается, что нужное приближение получено с заданной точностью ε (эпсилон), если вычислена сумма нескольких первых слагаемых, и очередное слагаемое оказалось по модулю меньше, чем данное малое положительное число ε. Тогда считается, что это и все последующие слагаемые можно уже не учитывать.

На каждом следующем шаге цикла будем максимально использовать сделанное нами на предыдущих шагах.

Если уже получено $x^{i-1}/(i-1)!$, то для вычисления $x^i/i!$ достаточно умножить предыдущий результат на x/i.

Блок-схема алгоритма решения задачи приведена на рис. 7.2.

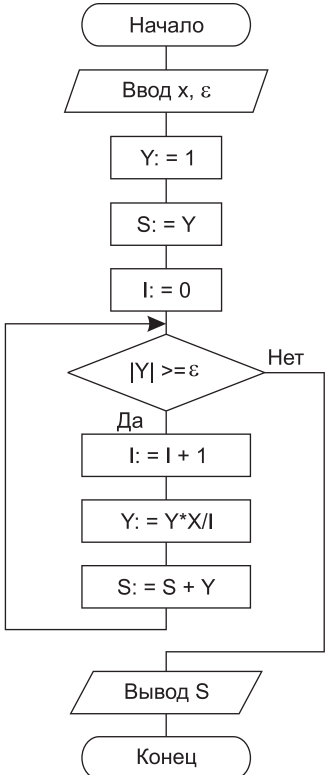


Рис. 7.2. Блок-схема алгоритма приближенного вычисления суммы ряда (пример 7.1)

Пример 7.1. Приближенное вычисление суммы бесконечно убывающего ряда

```
Program Summer2;
var
   Eps:real;     { Заданное число "эпсилон" }

   X: real;      { Основание степени в числителе дроби }

   S: real;      { В этой переменной будем
                   накапливать сумму }

   Y: real;      { Для хранения очередного слагаемого }

   i: integer;   { Счетчик числа шагов }
begin
   write('Введите X и Epsilon:');
   readln(X, Eps);

   Y:=1;         { Первое слагаемое }

   S:=Y;         { Положим в сумматор первое слагаемое }

   i:=0;         { Обнулим счетчик шагов }

   while abs(Y)>=Eps do { Пока добавленное слагаемое
                          не меньше "Эпсилон",
                          считаем сумму.
                          Если "эпсилон" сразу
                          не меньше 1,
                          цикл не выполнится ни разу! }

   begin         { Началось тело цикла }

      inc(i);    { Вычислили номер текущего шага }

      Y:=Y*X/i;  { Посчитали новое слагаемое }

      S:=S+Y     { Увеличили сумму на текущее слагаемое }

   end;          { Конец описания тела цикла.
                   После этой строки компьютер перейдет
                   на оператор while для сравнения
                   переменной "эпсилон" с только
                   что добавленным слагаемым }
```

```
   { Теперь выведем результат на экран }
   writeln('Сумма чисел=', S:6:4);
   readln
end.
```

Задание 7.1. Вычислить сумму ряда:

$$\frac{1}{1^2} + \frac{1}{3^2} + \frac{1}{5^2} + \frac{1}{7^2} + \dots$$

Вычисления прекращаются по тому же условию, что и в примере 7.1.

Полученная сумма должна быть близка к числу $\pi^2/8$.

Возведение числа в указанную целую степень

Рассмотрим задачу: возвести число *a*, введенное с клавиатуры, в степень *n*.

Задачу будем выполнять за $n + 1$ шаг.

Например, возведем число 2 в степень 3 (2^3):

0 шаг: $2^0 = 1$

1 шаг: $2^1 = 2^0 \cdot 2$ $(1 \cdot 2)$

2 шаг: $2^2 = 2^1 \cdot 2$ $(2 \cdot 2)$

3 шаг: $2^3 = 2^2 \cdot 2$ $(4 \cdot 2)$

Пример 7.2. Возведение числа a, введенного с клавиатуры, в степень n

```
Program Stp;
var P: real;    { Переменная, которая хранит результат
                  очередного шага }

   N: integer;  { Показатель степени }

   i: integer;  { Счетчик числа шагов }

   A: real;     { Основание степени }
begin
   write('Введите основание степени:');
   readln(A);
   write('Введите показатель степени:');
   readln(N);

   i:=0;        { 0-й шаг }
```

```
  P:=1;         { 2⁰=1 }

  while i<abs(N) do   { Показатель может быть
                          отрицательным, поэтому
                          используем для анализа
                          его абсолютную величину.
                          Если показатель N=0,
                          то в тело цикла
                          не попадаем ни разу,
                          так как 0-й шаг уже сделан }
  begin
     inc(i);    { Увеличиваем i на 1,
                  то есть i теперь равно номеру
                  текущего шага }

     P:=P*A     { Получаем результат i-го шага,
                  то есть A^i }
  end;

  { В переменной P на данный момент получен результат
    для положительного N }
  if N<0 then  { Если показатель N — отрицательный, }

     P:=1/P;    { то результат должен иметь
                  обратную величину }

  writeln('Результат=',P:6:3);
  readln
end.
```

Задание 7.2. Используя цикл с предусловием, написать программу вычисления $N!$.

Задание 7.3. Выполните задачу из предыдущей темы (задание 6.8), но используйте для этого цикл с предусловием. Блок-схема алгоритма вывода звездочек в верхней (2-й) строке с 3-го столбца (координата x) до 75-го столбца приведена на рис. 7.3. Продолжите блок-схему. Будьте внимательны с условиями!

Обратите внимание: нам понадобилось самим устанавливать значение x до входа в цикл и увеличивать x на 1 в теле цикла! В цикле со счетчиком это все делалось за нас в самой конструкции цикла.

Задание 7.4. Измените в задании 7.3 в теле цикла шаг счетчика, сделав его равным 3.

Задание 7.5. Проведите звездочки по диагонали из нижнего левого угла в верхний правый угол. Сначала заполните блок-схему алгоритма (рис. 7.4).

Пояснение: координата x изменяется быстрее, чем y, поскольку экран прямоугольный.

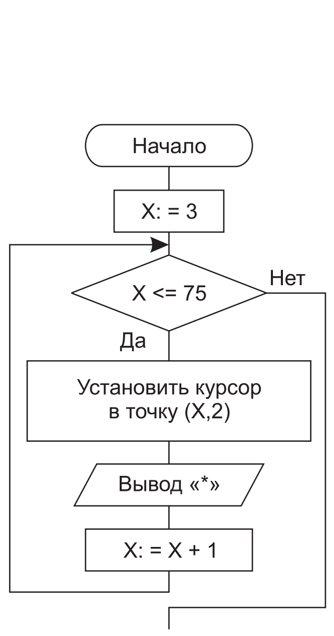


Рис. 7.3. Блок-схема алгоритма вывода звездочек во 2-й строке экрана с 3-й по 75-ю позицию

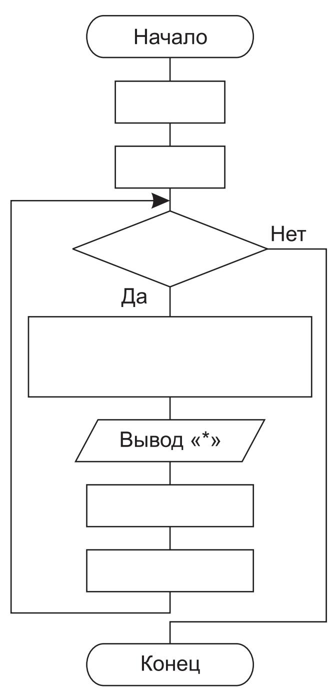


Рис. 7.4. «Слепая» блок-схема алгоритма вывода звездочек по диагонали из левого нижнего в правый верхний угол

Задание 7.6. Выведите в центр экрана с задержкой друг относительно друга следующие числа: 1, 2, 4, 8, 16, 32, 64, 128, 256.

Вероятно, вы поняли, что это степени числа 2.

Заполните блок-схему алгоритма (рис. 7.5), затем напишите программу.

Задание 7.7. Введите два числа (например, A = 5 и B = 8) и найдите их произведение, используя только операцию сложения. Заполните блок-схему алгоритма (рис. 7.6).

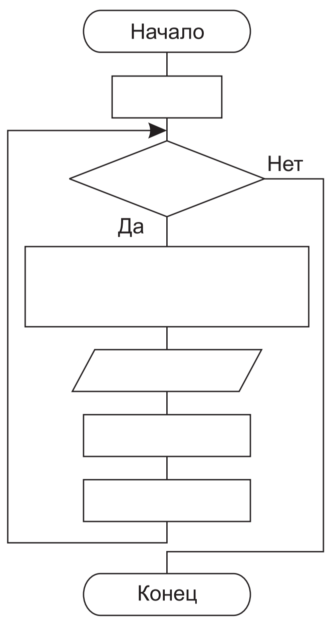


Рис. 7.5. «Слепая» блок-схема алгоритма вывода в центр экрана степеней числа 2 с временной задержкой

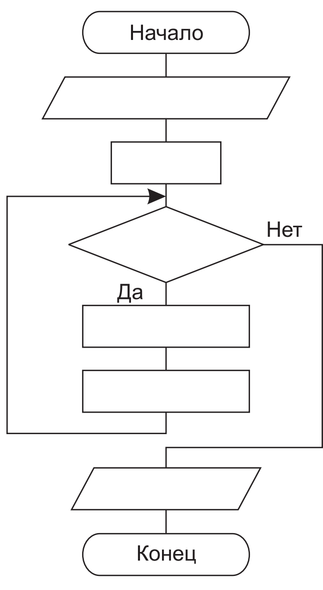


Рис. 7.6. «Слепая» блок-схема алгоритма вычисления произведения двух чисел с использованием только операции сложения

Задание 7.8. Введите два числа (например, A = 45 и B = 8) и найдите частное от деления нацело первого числа на второе (A на B) (в переменной k) и остаток от деления нацело (в переменной A), используя только операцию вычитания. Заполните блок-схему алгоритма (рис. 7.7).

Пояснение: в переменной k подсчитывайте, сколько раз сделана операция вычитания, то есть сколько раз число B содержится в числе A.

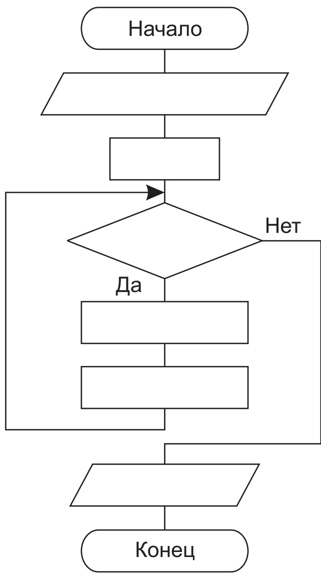


Рис. 7.7. «Слепая» блок-схема алгоритма вычисления целого частного и остатка от деления одного числа на другое с использованием только операции вычитания

Урок 7.2. Цикл с постусловием

Вторая разновидность цикла проверяет условие после выполнения тела цикла. Поэтому правильно будет назвать это условие *условием окончания цикла*. Цикл такого вида называется *циклом с постусловием*.

Цикл будет повторяться до тех пор, пока проверка этого условия будет давать результат «ложь» (false), то есть пока условие не выполнено. Даже если условие сразу окажется истинным, цикл выполнится хотя бы один раз.

Описание цикла с постусловием

Блок-схема в общем виде выглядит так (рис. 7.8).

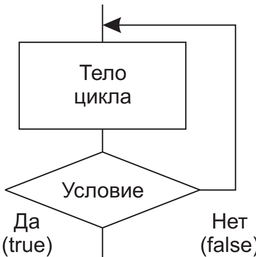


Рис. 7.8. Блок-схема цикла с постусловием

Выполнение цикла продолжается, если проверка логического условия дает результат «ложь». Если логическое условие выполняется, то происходит выход из цикла. Иными словами, если в цикле `while` проверялось условие продолжения цикла, то в цикле `repeat … until` — условие окончания.

На языке Паскаль этот тип цикла реализуется так:

```
repeat
   <тело цикла>
{ операторы begin ... end не требуются! }
until <логическое условие>
```

Использование циклов repeat и while

Рассмотрим задачу, в которой требуется вводить с клавиатуры числа и подсчитывать их сумму. Сумму необходимо подсчитывать до первого введенного отрицательного числа. Блок-схема алгоритма приведена на рис. 7.9.

Пример 7.3. Использование цикла repeat для подсчета суммы вводимых чисел до первого отрицательного числа

```
Program Summer1;
var
   sum, a: real;   { sum — для накопления суммы,
                     a — для очередного числа }
begin
   sum:=0;         { Обнуляем сумму }
```

```
  a:=0;          { Это тактическая хитрость
                   (см. замечание к примеру) }

  repeat

    sum:=sum+a;  { Добавляем введенное число к сумме }
    write('Введите число:'); { Ввод очередного числа }

    readln(a)

  until a<0;     { Проверяем введенное число
                   на отрицательность }

  { При выходе из цикла выполняется этот оператор: }
  writeln('Сумма чисел=', sum:5:3);
  readln
end.
```

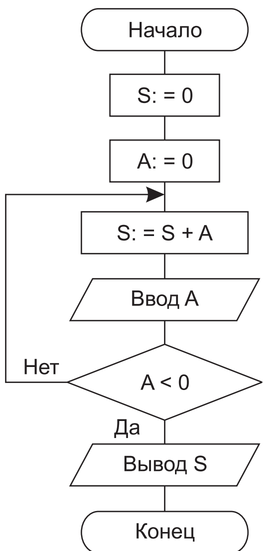


Рис. 7.9. Блок-схема алгоритма подсчета суммы вводимых элементов до первого отрицательного числа на базе цикла с постусловием

Использование оператора `repeat … until` оправдано тогда, когда нужны повторяющиеся действия, от выполнения которых зависит

дальнейшее продолжение цикла. Так, в приведенном примере продолжение цикла зависит от введенного числа. Если сразу введено отрицательное число, его не нужно добавлять к сумме. Если число неотрицательное, то нужно добавить его к сумме и продолжить выполнение цикла. Если перевести вышесказанное буквально на язык блок-схем, алгоритм должен выглядеть так (рис. 7.10):

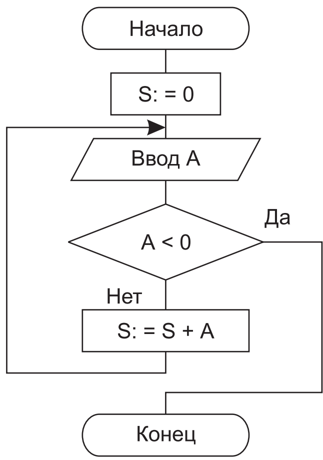


Рис. 7.10. Блок-схема алгоритма подсчета суммы вводимых элементов до первого отрицательного числа с выходом из цикла в середине тела цикла

Однако этот пример цикла нельзя отнести к циклам с пред- или постусловием. Здесь условие окончания цикла находится в середине. Хитрость примера 7.3 состоит в том, чтобы выполнить оператор `S:=S+A` до проверки условия окончания цикла и не нарушить правильности алгоритма. Ведь если поставить `S:=S+A` после ввода переменной `A`, то введенное отрицательное число добавится к сумме, чего быть не должно. А если поставить его перед вводом переменной `A`, то что же тогда прибавится к сумме во время первого шага цикла? Ведь переменная `A` еще не введена! Изначально присвоив переменной `A` нулевое значение, мы решаем эту проблему.

Вы скажете: если `repeat` здесь использовать неудобно, не лучше ли использовать `while`?

Давайте посмотрим, какую программу придется написать в этом случае, и сравним получившиеся два варианта:

Пример 7.4. Использование цикла `while` для подсчета суммы вводимых чисел до первого отрицательного числа

```
var
   sum, a: real;
begin
   sum:=0;                     { Обнуляем сумму }
   write('Введите число:');  { Ввод первого числа }

   readln(a);                  { Так как мы собираемся
                                 проверять переменную A
                                 до начала цикла,
                                 ей необходимо присвоить
                                 начальное значение }

   while a>=0 do               { Проверяем введенное число
                                 на отрицательность }

   begin
      sum:=sum+a;  { Добавляем введенное число к сумме }

      write('Введите число:'); { Ввод очередного числа }

      readln(a)
   end;

   { При выходе из цикла выполняется этот оператор: }
   writeln('Сумма чисел=', sum:5:3);
   readln
end.
```

Необходимость задать начальное значение переменной `A` вынуждает нас повторить операторы ввода переменной `A` дважды — до цикла и внутри него. С этой точки зрения использование `while` оказывается менее удобным.

Относительность выбора операторов while и repeat

Со временем вы поймете, что проверка условия окончания цикла до или после тела цикла — это вопрос исключительно личных предпочтений. В данном случае, например, можно написать эту программу через `while` и при этом не повторять оператор ввода:

Пример 7.5. Использование цикла while для подсчета суммы вводимых чисел до первого отрицательного числа без дублирования оператора ввода

```
var
   sum, a: real;
begin
   sum:=0;           { Обнуляем сумму }
   a:=1;             { Используем ту же хитрость,
                       что и в примере 7.3
                       Это нужно, чтобы значение A
                       удовлетворяло условию while }

   while a>=0 do     { Проверяем введенное число
                       на отрицательность }

   begin
      write('Введите число:'); { Ввод очередного числа }

      readln(a);

      sum:=sum+a     { Добавляем введенное число к сумме }
   end;
   writeln('Сумма чисел=', sum:5:3);
   readln
end.
```

В приведенных примерах должно быть хорошо видно, насколько важен порядок выполнения действий внутри цикла. Достаточно переставить местами операторы — и программа начинает работать совершенно иначе. Для начинающих это самое трудное. Даже если понятно, какие операторы должны выполняться в теле цикла, — то как определить, в правильном ли порядке мы их расставили?

Мы рекомендуем не лениться и всегда использовать один волшебный метод — трассировку! Честно и аккуратно выполнив вручную несколько шагов цикла, можно понять, правильно ли написана программа.

Задание 7.9. Написать программу, которая подсчитывает произведение целых чисел, введенных с клавиатуры. Произведение подсчитывается до тех пор, пока вводятся числа в интервале от –10 до +10. Используйте цикл с постусловием.

Подсказка: в записи условия используйте логическую операцию.

Задание 7.10. Выполните ту же задачу, но с использованием цикла с предусловием.

Задание 7.11. Написать программу «Угадай-ка»:

С использованием датчика случайных чисел в программе загадывается число в диапазоне 0...100. На отгадывание числа дается 10 попыток. Играющий вводит каждый раз очередное число. После каждого ответа программа выводит на экран одно из сообщений — `больше`, `меньше` или `угадано`, в зависимости от числа, введенного пользователем. Цикл завершается при выполнении одного из двух условий: либо число попыток достигло 10, либо дан правильный ответ.

Подсказки:

1. Каждый раз в цикле наращивается переменная `k`, содержащая счетчик попыток.
2. Для отслеживания правильного ответа введите логическую переменную `flag`, которой первоначально следует присвоить значение `False`. Если ответ верен, присвойте этой логической переменной значение `True`.
3. Цикл завершается, если значение счетчика попыток `k` равно 10 или если логическая переменная имеет значение `True` (использовать операцию логическое «или» — `or`).
4. При выходе из цикла надо сообщить, угадано ли число или же выход из цикла произошел по совершении 10 попыток. Для этого надо проверить значение логической переменной `flag` («истина» или «ложь»).

   Сначала заполните блок-схему алгоритма (рис. 7.11).

Задание 7.12. Введите два числа (например, `A` = 5 и `B` = 8) и найдите их произведение, используя только операцию сложения. Нарисуйте блок-схему алгоритма, используя цикл с постусловием.

Задание 7.13. Введите два числа (например, `A` = 45 и `B` = 8) и найдите частное от деления нацело (в переменной `k`) и остаток

от деления нацело (в переменной A), используя только операцию вычитания. Нарисуйте блок-схему алгоритма, используя цикл с постусловием.

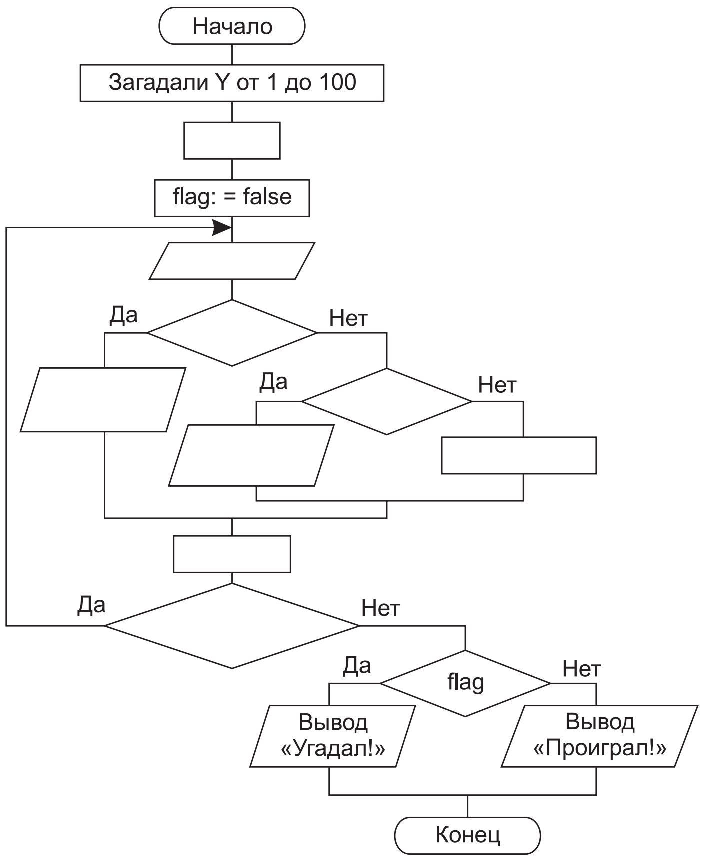


Рис. 7.11. «Слепая» блок-схема алгоритма «Угадай-ка» (см. задание 7.11)

Задание 7.14. Перед вами блок-схема алгоритма подсчета количества десятичных разрядов в заданном положительном числе N (рис. 7.12). Заполните таблицу трассировки и докажите, что в переменной k мы действительно получаем количество разрядов (в нашем случае их 4).

Таблица 7.1. Таблица для трассировки алгоритма, приведенного на рис. 7.12 (задание 7.14)

Оператор	Условие	N	K	Примечание
Ввод N		1024		

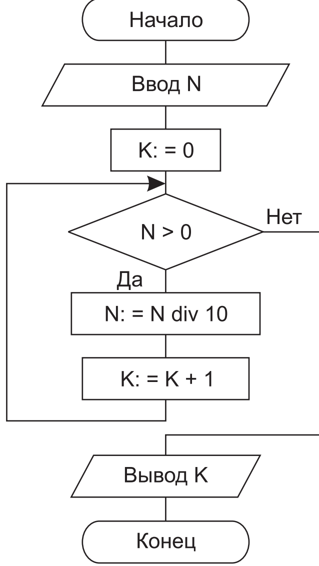


Рис. 7.12. Блок-схема алгоритма подсчета количества десятичных разрядов в заданном положительном числе N

Напишите программу и проверьте алгоритм для других значений N.

Задание 7.15. Перед вами блок-схема алгоритма подсчета суммы десятичных разрядов в заданном положительном числе N (рис. 7.13). Заполните таблицу трассировки и докажите, что в переменной S мы действительно получаем сумму разрядов (в нашем случае сумма равна 14).

Таблица 7.1. Таблица для трассировки алгоритма, приведенного на рис. 7.13 (задание 7.15)

Оператор	Условие	N	K	Примечание
Ввод N		4235		

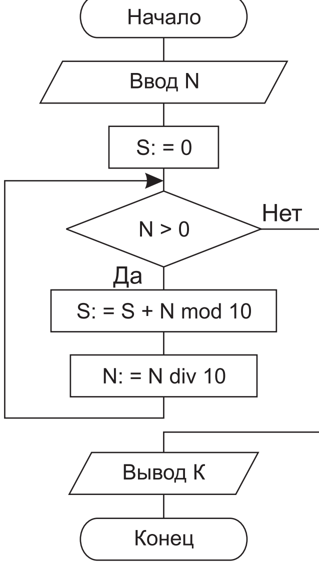


Рис. 7.13. Блок-схема алгоритма подсчета суммы десятичных разрядов в заданном положительном числе N

Напишите программу и проверьте алгоритм для других значений `N`.

Полезным примером цикла `repeat … until` является использование библиотечной функции `KeyPressed`.

`KeyPressed` — библиотечная функция (из модуля Crt), которая, отработав, возвращает в программу результат в виде логиче-

ского значения. Первоначально результат равен False. Как только будет нажата любая клавиша на клавиатуре, возвращаемый результат станет равным True. Таким образом, цикл вида repeat until KeyPressed будет выполняться, пока не будет нажата какая-либо клавиша.

Обычно это используется для получения задержки в программе — например, для просмотра результата на экране. Тело цикла пусто, цикл сводится только к проверке условия KeyPressed.

Выводы

1. Для организации многократно повторяющихся действий с неизвестным числом повторений используется оператор цикла с предусловием:
   ```
   while <логическое условие> do <оператор>
   ```
2. Выполнение цикла while прекращается, как только логическое условие примет значение false.
3. Цикл с предусловием (while) может не выполниться ни разу.
4. Для выполнения той же задачи служит цикл с постусловием:
   ```
   repeat
   <один или несколько операторов>
   until <логическое условие>
   ```
5. Выполнение цикла repeat прекращается, как только логическое условие примет значение true.
6. Цикл с постусловием (repeat) будет выполнен хотя бы один раз.
7. При использовании нескольких операторов в теле цикла repeat … until операторная скобка (begin … end) не нужна, так как пара repeat … until сама является операторной скобкой.

Контрольные вопросы

1. В каких случаях предпочтительнее использовать оператор цикла for, а в каких — операторы цикла с условием?
2. Чем проверка условия выполнения цикла while отличается от проверки в цикле repeat … until?

3. Что будет на экране в результате выполнения следующих фрагментов программ? (Переменные описаны для всех фрагментов программ.)

```
var
  k,x,i: integer;
begin

  { Фрагмент 1 }
  k:=256;
  while k<>1 do
    begin
      write(k:4);
      k:=k div 2
    end;

  { Фрагмент 2 }
  k:=5;
  repeat
    writeln(k);
    k:=k+5
  until k>30;

  { Фрагмент 3 }
  for i:=1 to 5 do
    writeln(i*5);

  { Фрагмент 4 }
  for i:=8 downto 4 do
    write(i:3);

  { Фрагмент 5 }
  for x:=1 to 79 do
  begin
    gotoxy(x,8);
    write('*');
    gotoxy(x,15);
    write('*')
  end;
end.
```

ТЕМА 8

Массивы — структурированный тип данных

В предыдущих уроках мы с вами рассматривали задачи, в которых использовалось небольшое количество данных. Для каждого мы создавали отдельную ячейку памяти с уникальным именем. Однако зачастую приходится работать с большим количеством однотипных данных. Для каждого из них требуется отдельная ячейка памяти — это понятно и естественно (два литра воды не поместятся в один стакан). А вот указывать для каждой ячейки отдельное имя — неудобно. Как быть?

Один из методов решения такой: выделим для этих данных область последовательных ячеек памяти и назовем всю область общим именем. А для того, чтобы к каждой ячейке можно было обратиться, пронумеруем ячейки по порядку. Таким образом, для обращения к определенной ячейке нужно указать название всей конструкции и номер ячейки в ней.

Урок 8.1. Хранение однотипных данных в виде таблицы

Массив — совокупность однотипных данных, хранящихся в последовательных ячейках памяти и имеющих общее имя. Ячейки называются *элементами массива*. Все элементы пронумерованы по порядку, и этот номер называется *индексом элемента массива*.

Все элементы массива имеют один и тот же тип. Сам массив при этом имеет имя — одно для всех элементов. Для обращения к конкретному элементу массива необходимо указать имя массива и (в квадратных скобках) индекс элемента.

Простейший вид массива — одномерный массив (рис. 8.1).

A	10	3	–8	14	25	12	10	1
	1	2	3	4	5	6	7	8

Рис. 8.1. Изображение одномерного массива в виде строки

`A` — имя массива, числа в клетках таблицы — элементы массива.

Рассмотрим запись `A[3] = -8`. В этой записи:

- `A` — имя массива;
- `3` — номер элемента массива (индекс);
- `A[3]` — обозначение 3-го элемента массива;
- `-8` — значение 3-го элемента массива.

Основные действия по работе с массивами

Нам предстоит научиться выполнять ряд наиболее распространенных действий с массивами:

- описание;
- заполнение массива случайными числами;
- заполнение массива с клавиатуры;
- вывод на экран;
- поиск максимального элемента;
- вычисление суммы всех элементов массива;
- вычисление количества положительных элементов.

Описание массива на языке Паскаль

```
<Имя массива>: array [<тип индекса>] of <тип компонентов>;
```

Здесь `<тип компонентов>` — это тип данных, который имеет каждый элемент массива, а `<тип индекса>` — границы изменения индекса.

Например:

```
var A: array [1..10] of integer;
```

Здесь тип индекса — интервальный, изменяется в интервале от 1 до 10, тип данных (элементов массива) — целый.

Заполнение массива случайными числами и вывод массива на экран

Рассмотрим задачу, в которой требуется с помощью датчика случайных чисел создать одномерный массив и вывести его на экран. Блок-схема алгоритма показана на рис. 8.2.

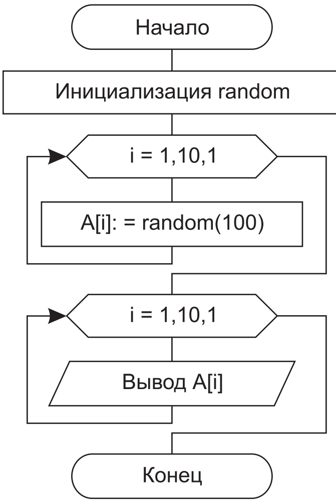


Рис. 8.2. Блок-схема алгоритма заполнения одномерного массива случайными числами и вывода массива на экран

Пример 8.1. Основные действия по работе с массивами

```
Program Massiv1;
uses Crt;
const      { Раздел описания констант,
             то есть постоянных величин,
             определяемых в программе заранее
             и не изменяющихся
             по ходу выполнения программы }

   N=10;   { Имена констант не используются для
             имен переменных величин (из раздела var) }
var
   A: array [1..N] of integer; { 1..N — тип индекса.
                                 Для индекса выбран
                                 интервальный тип,
                                 то есть интервал целых
```

```
                                    чисел от 1 до N,
                                    где N определено
                                    в разделе const }

   i:integer;   { Переменная, хранящая индекс элемента
                  массива, к которому идет обращение }
begin

{ II. Задание значений элементов массива
      как случайных чисел }

   Randomize; { Инициализация датчика случайных чисел }

   { Задание элементов массива: }

   for i:=1 to N do        { Переменная i изменяется
                             в цикле от 1 до N,
                             то есть мы по очереди
                             перебираем
                             все элементы массива }

      A[i]:=Random(100);   { В очередной элемент массива
                             A[i] записываем
                             случайное число от 0 до 99.
                             обратите внимание: i - номер
                             элемента массива (принято
                             говорить "индекс"), A[i] -
                             значение элемента массива }

{ III. Вывод элементов массива на экран в одну строку }
   ClrScr;
   writeln('Введенный массив:');
   for i:=1 to N do
      write(A[i]:4);       { На каждый элемент массива
                             выделяется по 4 позиции
                             строки, чтобы они
                             не склеивались при выводе! }

   writeln;                { Этот "пустой" оператор вывода
                             отработает только один раз
                             и переведет  курсор
                             на новую строку
                             для дальнейшей работы }
   readln
end.
```

В данном примере мы заполнили массив случайными числами от 0 до 99. Это обеспечила нам функция `random(100)`. А если нам нужно получить случайные числа в другом диапазоне — например, не от нуля? Расчет нужно сделать такой: функция `random(N)` выдаст `N` различных чисел от 0 до `N` – 1. Если нам нужно, чтобы наименьшим числом моего диапазона было `K`, значит, необходимо прибавить это `K` к `random(N)`. Наибольшее число, которое будет выдавать в этом случае формула `random(N) + K`, будет наибольшим числом диапазона.

Пусть, например, нам требуются случайные числа в диапазоне –100..+100. Считаем, сколько различных чисел в этом диапазоне: 100 положительных, 100 отрицательных и ноль. Итого 201. Формула тут проста: вычесть из большего меньшее и прибавить 1. Значит, `N` = 201, а `K` = –100. То есть получаем формулу `random(201) - 100`.

ЗАМЕЧАНИЕ

К сожалению, в таком виде формула работать не будет — при запуске программа вылетит с сообщением об ошибке. Это от «излишнего ума», который проявляет здесь среда Turbo Pascal. Дело в том, что Turbo Pascal считает тип этого выражения по функции `random`. *А она имеет тип* `word`. *Иными словами, беззнаковый. При попытке вычесть 100 из числа, меньшего 100, получаем отрицательный результат, что Turbo Pascal не устраивает. Самый простой способ обойти эту напасть — поменять местами уменьшаемое (*`random`*) и вычитаемое, то есть написать* `-100 + random(201)`. *Тогда Turbo Pascal будет считать тип этого выражения как* `integer` *по первому числу (*`-100`*), и ошибки не возникнет.*

Задание 8.1. Оформите эту программу так, чтобы задание массива и вывод его элементов на экран выполнялись в одном цикле. Вам понадобится составной оператор для тела цикла `begin … end`.

Задание 8.2. Добавьте в программу задания 8.1 новый цикл вывода элементов массива в обратном порядке (начиная с последнего). Попробуйте выполнить то же задание без введения нового цикла.

ЗАПОМНИТЕ!

Массив — это множество ячеек памяти. Поэтому любое действие с массивом состоит в том, чтобы перебрать все эти ячейки или, по крайней

мере, какую-то их часть. Это значит, что любое действие с массивами должно содержать в себе цикл, в котором перебираются элементы массива. Если вы пишете программу с массивом и не написали цикла (`for`*,* `while` *или* `repeat`*) — значит, вы ошиблись.*

Создание пользовательского типа данных

В следующем примере массив описывается в новом разделе — разделе описания типов пользователя (`type`). Вы можете по-прежнему пользоваться описанием массива в разделе описания переменных (как в примере 8.1). Вариант описания массива, приведенный в этом примере, в большей степени соответствует грамотному стилю оформления программы.

Пример 8.2. Ввод с клавиатуры одномерного массива целых чисел и вывод его элементов на экран с противоположным знаком

```
Program Massiv2;
const
   N=10;

type           { Раздел описания типов переменных.
                 Эти типы определяет сам пользователь,
                 то есть мы определяем тип одномерного
                 массива из n целых чисел }

   Mas=array [1..N] of integer;   { 1..N — тип индекса;
                 для индекса выбран интервальный тип,
                 то есть интервал целых чисел от 1 до N,
                 где N определено в разделе const }

var
   Line:Mas;   { Line — одномерный массив,
                 его тип определен нами как Mas }

   i:integer;  { Переменная, хранящая индекс элемента
                 массива, к которому идет обращение }

begin

{ IV. Ввод массива с клавиатуры }

   for i:=1 to N do   { Обращение к элементам массива
                        происходит в цикле, по очереди }
```

```
    begin      { Начало цикла ввода элементов массива }
       write('Введите элемент с индексом ',i,':');

       readln(Line[i]) { Обращаемся к i-му
                         элементу массива
                         (Line[1],Line[2] и т. д.) }

    end;       { Конец цикла ввода элементов массива }

    { Вывод элементов происходит также в цикле:}

    for i:=1 to N do   { Перебираем все N элементов
                         массива }

       write(-Line[i]:5);   { 10 элементов выводятся
                              в строку. Выводим все
                              элементы массива
                              с противоположным знаком }

    writeln;        { После вывода массива элементов —
                      переход на новую строку }
    readln
 end.
```

Заметьте, в задании не требовалось, чтобы знак всех элементов массива менялся на противоположный. Требовалось лишь вывести их в таком виде на экран. Если бы требовалось изменить сам массив, обязательно нужно было бы сделать примерно следующее:

```
For i:=1 to N do
  Line[i]:=-Line[i];
```

Очень важно понимать: то, что мы видим на экране, не всегда есть то, что действительно хранится в памяти. Программист может играть здесь роль фокусника. Важно лишь, чтобы задание было точно выполнено.

То есть, если в задании требуется изменить каким-либо образом данные и вывести результат на экран, а программист просто вывел данные на экран в нужном виде, то такое задание не может считаться выполненным.

Например, в задании требуется поменять все элементы массива в обратном порядке и вывести результат на экран. Программист

решил, что проще всего будет просто вывести на экран все элементы, начиная с последнего. Такое решение не засчитывается!

Задание 8.3. Выполнить следующие действия:

1) создать одномерный массив A из 10 целых чисел (с помощью датчика случайных чисел);
2) вывести массив на экран в виде строки чисел;
3) подсчитать сумму элементов массива (блок-схема алгоритма показана на рис. 8.3);
4) вывести сумму на экран.

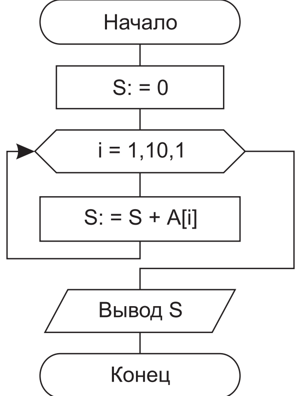


Рис. 8.3. Блок-схема алгоритма вычисления суммы элементов одномерного массива

Задание 8.4. Выполнить следующие действия:

1) создать одномерный массив A из 10 целых чисел;
2) вывести массив на экран в виде строки чисел;
3) поменять местами элементы массива (блок-схема алгоритма показана на рис. 8.4) следующим образом:

   1-й элемент — со 2-м;
   3-й — с 4-м;
   5-й — с 6-м;
   7-й — с 8-м;
   9-й — с 10-м
   (надо вспомнить, как идет обмен значениями двух переменных (рис. 2.4));

4) вывести измененный массив на экран.

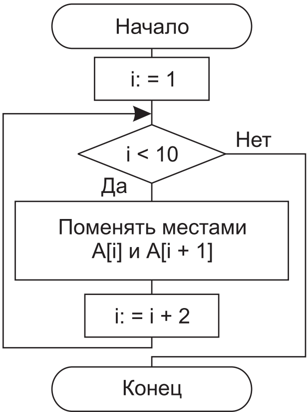


Рис. 8.4. Блок-схема алгоритма обмена соседних элементов одномерного массива

Поиск максимального элемента массива

Довольно-таки типичная задача для большого количества данных — поиск максимума. Например, в списке успеваемости учеников класса найти самого прилежного. Иначе говоря, требуется выбрать наибольшее значение среднего балла и указать фамилию ученика.

Пример 8.3. Программа поиска максимального элемента в массиве и его индекса (см. блок-схему алгоритма на рис. 8.5)

```
Program Maximum;
const
   N=10;
type
   Mas=array [1..N] of integer;
var
   A:Mas;

   i   :integer;     { Счетчик цикла }

   Max :integer;     { Переменная для хранения
                       величины максимального элемента }
```

```
   Imax:integer;     { Переменная для хранения
                       индекса максимального элемента }

begin                { Тело программы }
   { Заполним элементы массива значениями датчика
     случайных чисел и выведем весь полученный массив
     на экран в одном цикле }
   Randomize;
   for i := 1 to N do
   begin
      A[i]:=-50+Random(101);
      write(A[i]:5)
   end;
   writeln;

{ V. Поиск максимального элемента
     и его индекса в массиве }

   Imax:=1;          { Сначала считаем,
                       что первый элемент массива
                       и есть максимальный }

   Max:=A[1]; { Его индекс и величину
                записываем соответственно
                в переменные Imax и Max }

   for i := 2 to N do   { Сравним нашего кандидата
                          в максимумы со всеми
                          остальными элементами массива
                          (со второго до последнего) }

      if Max < A[i] then { Если наш кандидат
                           в максимумы оказался
                           меньше текущего элемента... }

      begin
         Max:=A[i];       { ...то будем считать теперь
                            кандидатом в максимумы
                            текущий элемент }

         Imax:=i          { Запомним его значение
                            и индекс
                            в переменных Max и Imax }
```

```
      end;
    writeln('Максимальный элемент в массиве=',Max:5);
    writeln('Его индекс=',Imax:5);
    readln
end.
```

Заметим, что в процессе поиска максимума не обязательно хранить обе величины — номер максимума и его значение. Достаточно хранить одну, в зависимости от поставленной задачи.

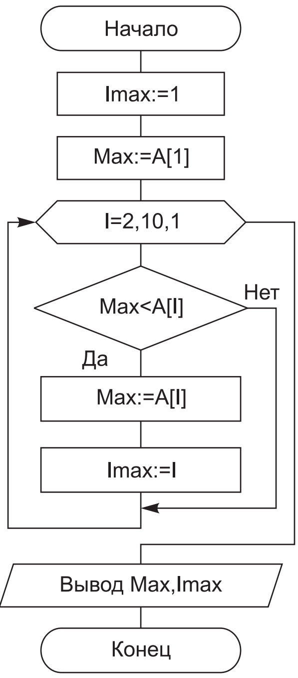


Рис. 8.5. Блок-схема алгоритма поиска максимального элемента массива и его индекса

Если индекс максимума не нужно знать, достаточно будет переменной `Max`. Если, наоборот, нужен только номер — достаточно `Imax`. Тонкость состоит в том, что если нужно найти и то и другое, все равно достаточно найти только `Imax`, ведь значение максимума легко может быть получено по его индексу (`A[Imax]`).

Иными словами, нашу программу можно упростить следующим образом:

Пример 8.4. Программа поиска максимума, не хранящая значение максимума, а запоминающая только его номер

```
Program Maximum2;
const
   N=10;
type
   Mas=array [1..N] of integer;
var
   A         :Mas;
   i, imax   :integer;
begin
   Randomize;
   for i := 1 to N do
   begin
      A[i]:=-50+Random(101);
      write(A[i]:5)
   end;
   writeln;

{ V. Поиск индекса максимального элемента в массиве }
   imax:=1;
   for i := 2 to N do
      if A[imax] < A[i] then
         imax:=i;
   writeln('Максимальный элемент в массиве=',A[imax]:5);
   writeln('Его индекс=',imax:5);
   readln
end.
```

Задание 8.5. Выполните поиск максимального и минимального элементов в массиве за один цикл (блок-схема алгоритма показана на рис. 8.6).

Задание 8.6. В одномерном массиве из 10 элементов определить местоположение минимального элемента. Обнулить элементы, стоящие до него, но не сам этот элемент. (Обнулить — значит, записать 0 на место элемента, то есть выполнить `A[i] := 0`.) Измененный массив вывести на экран.

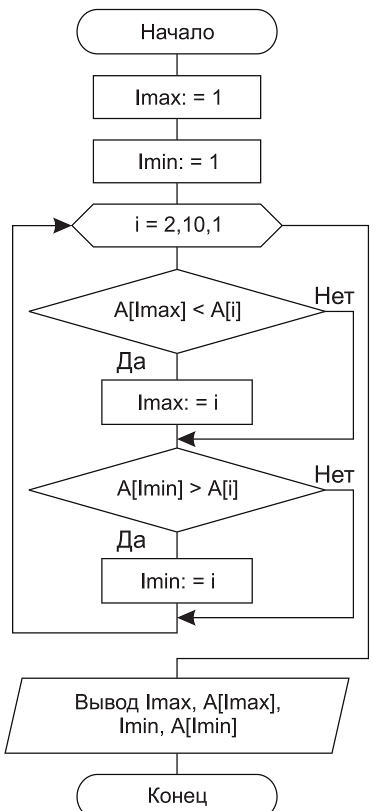


Рис. 8.6. Блок-схема алгоритма поиска индексов максимального и минимального элементов массива за один цикл

Задание 8.7. В одномерном массиве из 10 элементов определить местоположение минимального и максимального элементов. Обнулить элементы, стоящие между ними, а также сами эти элементы.

Вычисление суммы и количества элементов массива с заданными свойствами

Еще одна задача — посчитать сумму элементов, которые удовлетворяют какому-то условию. Наверное, вы уже поняли общий принцип: перебираем все элементы массива (цикл `for`) и про-

веряем для каждого элемента выполнение условия (оператор `if`). Если условие выполнено, добавим элемент к сумме (`S := S+A[i]`).

Пример 8.5. Вычисление суммы положительных элементов массива

```
program PositivSumm;
const   N=10;
type    Mas=array [1..N] of integer;
var a:Mas;
    i:integer;     { Счетчик цикла }

    S:integer;     { Копилка — переменная
                     для суммирования
                     положительных элементов }
begin
   { Заполним массив случайными числами
     в диапазоне -100..+100 }
   randomize;
   for i:=1 to N do
   begin
      a[i]:=-100+random(201);
      write(a[i]:5)
   end;
   writeln;

   { Присвоим переменным начальные значения }

   S:=0;            { Переменная S — аккумулятор.
                     Она будет накапливать сумму
                     всех положительных элементов.
                     Нужно присвоить ей такое
                     начальное значение, чтобы оно
                     не повлияло на результат
                     суммирования. Таким числом
                     является ноль }

   for i:=1 to N do { Перебираем все элементы массива }

      if A[i]>0 then   { Проверяем каждый элемент
                         на положительность }

         S:=S+A[i];    { Если элемент положительный,
                         добавляем значение элемента
                         к аккумулятору }
```

```
  { Выводим результат на экран: }
   writeln('Сумма положительных элементов=',S);
   readln
end.
```

Вот задача, очень похожая на предыдущую: посчитать количество элементов массива с указанными свойствами. Решается аналогично: переберем все элементы, проверим для каждого условие и, если оно выполнено, увеличим счетчик на единицу (`k := k+1`).

Пример 8.6. Вычисление количества четных элементов массива (блок-схема алгоритма показана на рис. 8.7)

```
program EvenCount;
const   N=10;
type    Mas=array [1..N] of integer;
var a:Mas;
    i:integer;    { Счетчик цикла }

    K:integer;    { Копилка — переменная для подсчета
                    количества четных элементов }
begin
   { Заполним массив случайными числами
     в диапазоне +10..+100 }
   randomize;
   for i:=1 to N do
   begin
      a[i]:=+10+random(91);
      write(a[i]:5)
   end;
   writeln;

   { Присвоим переменным начальные значения }

   K:=0;            { Переменная K — счетчик.
                      Она будет выполнять ту же функцию,
                      что и пальцы на руке при счете:
                      каждый раз, когда будет встречаться
                      четное число, будем загибать один
                      палец, то есть увеличивать
                      переменную K на единицу.
```

```
                    В начале нужно присвоить ей
                    такое значение, чтобы оно
                    не повлияло на результат
                    суммирования. Таким числом
                    является ноль }

  for i:=1 to N do  { Перебираем все элементы массива }

     if a[i] mod 2 = 0 then { Проверяем каждый элемент
                               на четность, то есть
                               проверяем, что остаток
                               от деления этого элемента
                               массива на 2 равен нулю }
        K:=K+1;  { Если элемент четный, увеличиваем
                   счетчик на единицу }

 { Выводим результат на экран: }
  writeln('Количество четных элементов=',K);
  readln
end.
```

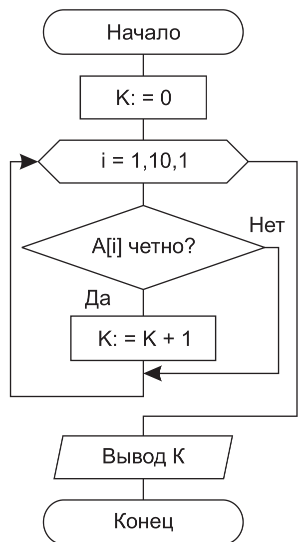


Рис. 8.7. Блок-схема алгоритма вычисления количества четных элементов массива

Урок 8.2. Поиск в массиве

Теперь рассмотрим еще ряд задач, которые приходится решать при работе с массивами, а именно задачи поиска. На примере задачи поиска отрицательного элемента мы рассмотрим несколько методов, применяемых для поиска в массиве.

Определение наличия в массиве отрицательного элемента с использованием флажка

Рассмотрим задачу: определить, есть ли в массиве хотя бы один отрицательный элемент.

На первый взгляд задача кажется весьма простой: достаточно перебрать все элементы и проверить каждый на отрицательность. Это правильно. Но вот что делать дальше? И как определить, что в массиве вообще нет отрицательных элементов? А если их несколько?

Данная задача может быть решена несколькими способами. Первый — самый простой — использовать флажок.

Пример 8.7. Определение наличия в массиве отрицательного элемента с использованием флажка

```
program Search1;
const  N=10;
type   mas=array [1..N] of integer;
var  a:mas;
     i:integer;
     fl:boolean;    { Флажок указывает
                      на успешность поиска }
begin
   { Заполним массив случайными числами }
   randomize;
   for i:=1 to N do
   begin
     a[i]:=-2+random(20);
     write(a[i]:4)
   end;
   writeln;

   { Начало блока поиска }
   { Первый вариант поиска отрицательного элемента —
     использование флажка }
```

```
    fl:=false;    { Изначально флажок = false,
                    так как мы еще ничего не нашли }

    for i:=1 to N do
       if a[i]<0 then { Проверяем каждый элемент
                        на отрицательность }

          fl:=true;   { Если нашли отрицательный,
                        устанавливаем флажок
                        в состояние "истина" }
    if fl then        { в конце проверяем флажок }
       writeln('В массиве есть отрицательный элемент')
    else
       writeln('В массиве нет отрицательных элементов');

    { Конец блока поиска }

    readln
end.
```

Определение наличия в массиве отрицательных элементов путем вычисления их количества

Нетрудно заметить, что использование логического флажка сужает функциональность метода. Ведь мы определяем только наличие отрицательного элемента, и ничего более. Если вместо флажка использовать целое число, оно сможет сообщить нам нечто большее о результатах поиска.

Пример 8.8. Определение наличия в массиве отрицательных элементов путем подсчета количества таких элементов

```
program Search2;
const  N=10;
type   mas=array [1..N] of integer;
var a:mas;
   i:integer;
   fl:integer; { Это уже не флажок, а счетчик количества
                 найденных отрицательных элементов }
begin
   { Заполним массив случайными числами }
   randomize;
   for i:=1 to N do
   begin
```

```
      a[i]:=-2+random(20);
      write(a[i]:4)
    end;
    writeln;

    { Начало блока поиска }
    { Второй вариант поиска отрицательного элемента —
      количество отрицательных элементов }

    fl:=0; { Изначально счетчик = 0,
             так как мы еще ничего не нашли }

    for i:=1 to N do
       if a[i]<0 then { Проверяем каждый элемент
                        на отрицательность }

          fl:=fl+1; { Если нашли отрицательный —
                      увеличиваем счетчик на единицу }

    writeln('Количество отрицательных элементов=',fl);
    if fl>0 then
       writeln('В массиве есть отрицательный элемент')
    else
       writeln('В массиве нет отрицательных элементов');

    { Конец блока поиска }

    readln
 end.
```

Нахождение номера отрицательного элемента массива

Мы только что рассмотрели перебор элементов массива в поисках элементов с какими-то свойствами (в нашем случае отрицательных). Можно также использовать этот метод для поиска номера отрицательного элемента.

Пример 8.9. Определение наличия в массиве отрицательного элемента путем вычисления его номера

```
program Search3;
const  N=10;
type   mas=array [1..N] of integer;
```

```
var a:mas;
    i:integer;
    fl:integer;     { Индекс найденного
                      отрицательного элемента }
begin
   { Заполним массив случайными числами }
   randomize;
   for i:=1 to N do
   begin
     a[i]:=-2+random(20);
     write(a[i]:4)
   end;
   writeln;

   { Начало блока поиска }
   { Третий вариант поиска отрицательного элемента –
     нахождение индекса отрицательного элемента }

   fl:=0; { Изначально флажок (индекс) = 0,
            так как мы еще ничего не нашли }

   for i:=1 to N do

      if a[i]<0 then  { Проверяем каждый элемент
                        на отрицательность }

         fl:=i;       { Если нашли отрицательный,
                        запоминаем его номер }

   { Теперь по значению переменной fl можно определить,
     был ли в массиве хоть один отрицательный элемент.
     Если fl остался равен нулю, значит проверка на
     отрицательность ни разу не выполнилась }

   if fl>0 then
      writeln('Индекс отрицательного элемента=',fl)
   else
      writeln('В массиве нет отрицательных элементов');

   { Конец блока поиска }

   readln
end.
```

Возможно, вы уже задаетесь вполне резонным вопросом: а если в массиве несколько отрицательных элементов, то какой из них мы нашли?

Так как при нахождении отрицательного элемента цикл не заканчивается, ответ очевиден: последний. Этот метод находит номер последнего в массиве отрицательного элемента.

А как нам найти первый отрицательный элемент?

Один из методов (на данный момент самый легкий) — исправить цикл так, чтобы он перебирал элементы с конца:

```
for k := N downto 1 do…
```

Теперь рассмотрим последнюю придирку к нашему алгоритму. Если нам нужно найти номер первого отрицательного элемента, то зачем перебирать все элементы? Достаточно остановиться, как только будет найден первый такой элемент.

Эти рассуждения подталкивают нас заменить цикл `for`, который перебирает все элементы, циклом `while`, который остановится в нужный нам момент:

```
k := 1;
while a[k] >= 0 do { Перебираем все не подходящие нам
                    (то есть неотрицательные) элементы }
 inc(k); { Берем номер следующего элемента }
```

Лирическое отступление. В только что приведенном примере мы использовали оператор inc, который раньше не упоминали. Он увеличивает значение указанной переменной на единицу. То есть оператор `inc(k)` аналогичен оператору `k := k + 1`. По аналогии с `inc`, в Turbo Pascal имеется еще оператор `dec`. Он уменьшает значение указанной переменной на единицу. Мы не привели эти операторы ранее, чтобы преждевременно не забивать ваши головы излишней информацией. Вы можете ими не пользоваться и продолжать писать `k := k + 1` и `k := k - 1`, как и ранее, вместо `inc(k)` и `dec(k)`.

Рассмотренный метод поиска первого отрицательного элемента обладает одним серьезным недостатком: он не останавливается, если в массиве вообще нет отрицательных элементов. Перебрав все элементы массива, цикл продолжит искать отрицательные элементы дальше. Это приведет или к зависанию компьютера, или к аварийному завершению программы в зависимости от настроек. Исправим ошибку. Для этого добавим еще одно условие окончания поиска — «если мы перебрали все элементы».

Пример 8.10. Определение наличия в массиве отрицательного элемента и вычисление его номера (если такой есть)

```
program Search4;
const  N=10;
type   mas=array [1..N] of integer;
var a:mas;
    i:integer;  { Здесь нам не нужен флажок –
                  счетчик цикла будет нашим флажком }
begin
   { Заполним массив случайными числами }
   randomize;
   for i:=1 to N do
   begin
     a[i]:=-2+random(20);
     write(a[i]:4)
   end;
   writeln;

   { Начало блока поиска }
   { Четвертый вариант поиска
     номера первого отрицательного элемента }

   i:=1;        { Так как используем цикл while вместо
                  for, приходится самим заботиться
                  о счетчике цикла }

   while (a[i]>=0) and (i<N) do { Перебираем все
                                  не подходящие нам
                                 (неотрицательные) элементы.
                                  Кроме того, проверяем
                                  выход за границы массива }

      inc(i);   { Берем номер следующего элемента }

   { Цикл закончен. Проверим, по какому из двух условий
     это произошло }

   if a[i]<0 then
      { Если элемент, на котором мы остановились,
        отрицательный - значит, мы его нашли.
        Иначе - нет отрицательных элементов }

      writeln('Индекс отрицательного элемента=',i)
```

```
  else
     writeln('В массиве нет отрицательных элементов');

  { Конец блока поиска }

  readln
end.
```

Задание 8.8. Определить, есть ли в массиве положительные четные элементы, и, если есть, вывести номер последнего из них.

Урок 8.3. Двумерные массивы

На предыдущих уроках мы рассмотрели с вами одномерные массивы. Это означает, что массивы имеют одно измерение — количество элементов. Визуально такие массивы можно представить как строку элементов. Однако наш мир не ограничивается одним измерением. На этом уроке мы рассмотрим массивы, которые можно визуально представить как таблицу.

Двумерный массив — это таблица из однотипных элементов, организованная по строкам и столбцам. Местоположение каждого элемента двумерного массива (матрицы) определяется индексом (номером) строки и индексом (номером) столбца (рис. 8.8).

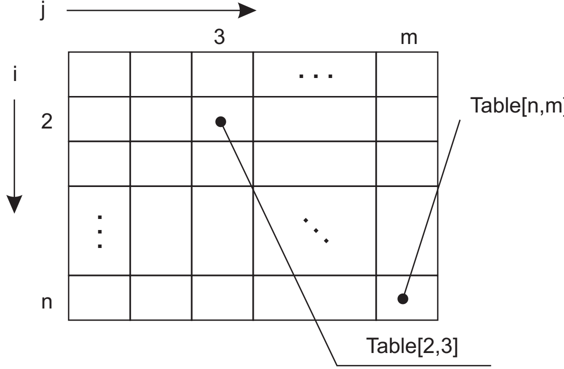


Рис. 8.8. Изображение двумерного массива в виде таблицы

В следующем примере мы создадим матрицу 5×4, задав значения ее элементов с помощью датчика случайных чисел, и выведем ее на экран по строкам. При этом будем использовать вложенные циклы: внешний цикл будет проходить по строкам, а внутренний — по столбцам. Блок-схема алгоритма представлена на рис. 8.9.

Пример 8.11. Создание матрицы 5×4, вывод ее на экран по строкам

```
program Massiv_2;
const
   N = 5;    { Число строк }
   M = 4;    { Число столбцов }
var
   Table : array [1..N, 1..M] of integer; { Заказываем
               область памяти для хранения двумерного
               массива из N строк и M столбцов }

{ Вообще говоря, нигде не определено,
  что первый индекс — это номер строки,
  а второй — это номер столбца.
  Так как выводом на экран занимается программист,
  он сам решает, как ему удобнее.
  Нам удобнее считать, что номер строки — первый индекс,
  а номер столбца — второй }

   i, j : integer;   { Переменные для хранения
                       индексов строки и столбца }
begin
   { Заполнение массива датчиком случайных чисел:}
   randomize;
   for i:=1 to N do
      for j:=1 to M do

         Table[i,j]:=Random(100); { Запись
                                   случайного числа
                                   в массив на место
                                   с номером строки i и
                                   номером столбца j }

   { Вывод матрицы на экран по строкам: }
   for i:=1 to N do
   begin
      for j:=1 to M do
```

```
        write(Table[i,j]);

      writeln          { Переход на новую строку
                         после вывода всех элементов
                         строки i }

    end;
    readln
  end.
```

Начало

K = 1,5,1

M = 1,4,1

Вывод A[K, M]

Переход на
след. строку

Конец

Рис. 8.9. Блок-схема алгоритма вывода двумерного массива 5×4 на экран по строкам;

Задание 8.9. Написать программу, в которой:

- определить матрицу 3×5;
- вывести ее на экран;
- определить величину максимального элемента данной матрицы и вывести на экран его значение и его позицию в матрице.

Выводы

1. Для хранения однотипных данных используется структурированный тип данных — массив (одномерный и многомерный).

2. Для описания массива необходимо указать его имя, тип данных (`array`), диапазон изменения индексов его элементов (в квадратных скобках) и тип элементов, из которых он состоит:

```
mas: array [1..20] of integer;
```

3. Обращение к каждому элементу массива идет по имени массива и по индексу (номеру) элемента в массиве.
4. При выполнении любых действий с массивами необходимо использовать циклы, в которых перебираются номера элементов массива.
5. При выполнении операций поиска в массиве нужно перебирать по очереди его элементы и проверять для каждого искомое условие.
6. Примером многомерного массива является двумерный массив (матрица). Обращение к элементам матрицы идет по ее имени, индексу строки и индексу столбца.

Контрольные вопросы

1. Какое условие должно выполняться, чтобы некоторое количество отдельных данных можно было объединить в один массив?
2. Что в записи `A[4] = -12` является именем массива, что — индексом, а что — значением элемента?
3. Чем одномерный массив отличается от двумерного?
4. Какие необходимы действия, чтобы вывести на экран все отрицательные элементы массива?
5. Почему при поиске какого-либо элемента в массиве нельзя обойтись без цикла?

ТЕМА 9

Вспомогательные алгоритмы. Процедуры и функции. Структурное программирование

В этой теме мы с вами рассмотрим методы декомпозиции. То есть мы будем учиться разбивать одну большую задачу на несколько отдельных, помельче. Небольшую самостоятельную задачу обычно гораздо легче решить. В этой теме мы познакомимся со способом записи таких подзадач на языке Паскаль.

Урок 9.1. Конструирование алгоритма «сверху вниз»

При конструировании достаточно сложного алгоритма логично разбивать его на ряд более простых задач. Построение алгоритма идет «сверху вниз». Сначала строится основной алгоритм. В нем записываются обращения к вспомогательным алгоритмам, которые позволят решить отдельные, более простые подзадачи (подзадачи 1-го уровня). Если есть необходимость, осуществляют дальнейшую детализацию, и подзадача разбивается на еще более простые задачи (подзадачи 2-го уровня). Вспомогательные алгоритмы для решения подзадач последнего уровня не содержат обращений к другим вспомогательным алгоритмам (рис. 9.1).

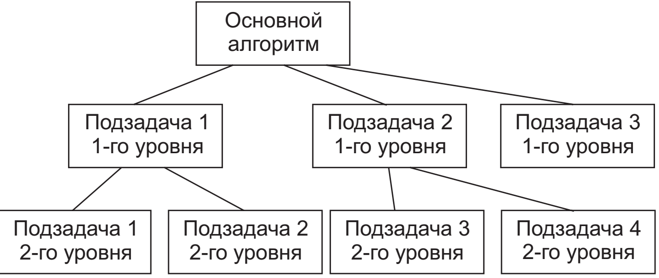


Рис. 9.1. Схема алгоритма, построенного по принципу «сверху вниз»

Итак, *вспомогательным алгоритмом* называется алгоритм решения некоторой задачи, являющейся подчиненной по отношению к исходной (основной) задаче. При реализации таких алгоритмов на языке Паскаль их оформляют в виде *процедур* или *функций*.

Согласно концепции структурного программирования вспомогательный алгоритм должен:

- иметь имя, по которому его можно вызвать из других алгоритмов;
- возвращать управление тому алгоритму, из которого он был вызван. После того как завершится выполнение вспомогательного алгоритма, вызвавший его алгоритм должен продолжить работу с той точки, в которой он был прерван;
- иметь возможность вызывать другие алгоритмы;
- иметь достаточно малые размеры.

Практическая задача с использованием вспомогательных алгоритмов

Задача: выполнить с массивом действия, которые были предложены в заданиях 8.3, 8.4 (см. урок 8.1):

а) заполнить одномерный целочисленный массив из 10 элементов случайными числами от –20 до +20;

б) вывести массив на экран в виде строки чисел;

в) подсчитать сумму элементов массива;

г) поменять местами элементы массива следующим образом:

1-й элемент — со 2-м,
3-й — с 4-м;
5-й — с 6-м;
7-й — с 8-м;
9-й — с 10-м;

д) вывести измененный массив на экран.

Разобьем основной алгоритм на подзадачи в порядке их перечисления в задании (рис. 9.2).

Обратим внимание: пункты б) и д) выполняют одинаковое действие — вывод массива на экран. Поэтому оформим их в виде отдельного фрагмента программы под некоторым именем, а в нужных местах вызовем этот фрагмент по его имени.

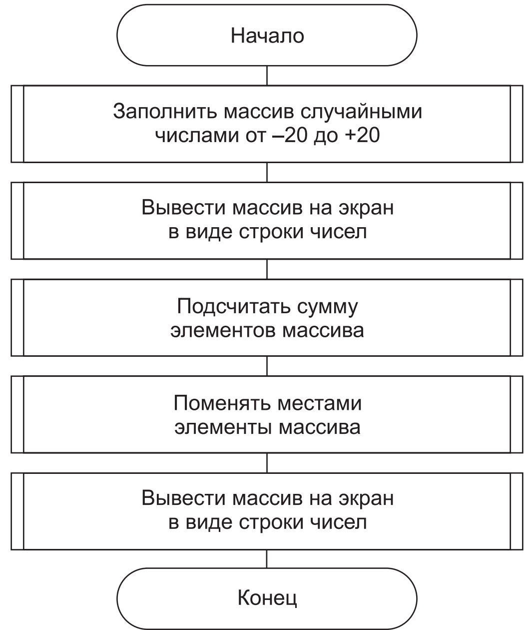


Рис. 9.2. Блок-схема алгоритма решения задачи из примера 9.1

Такой фрагмент называется *процедурой*.

Процедура оформляется по тем же правилам, что и программа.

Пункты а) и г) оформим также в виде процедур.

Пункт в) — вычисление суммы элементов массива — оформим отдельным фрагментом, но с возвращением результата вычисления. Такой фрагмент называется *функцией*. В отличие от процедуры, из функции всегда возвращается результат в виде переменной, тип которой указывается в заголовке функции (см. пример 9.1).

Такая организация программы по блокам вспомогательных алгоритмов (процедурам и функциям) придает ей логичную структуру и делает ее более удобной для дальнейшего использования.

Процедуры и функции в программе оформляют в виде отдельных блоков, каждый из которых начинается специальным словом (`procedure` или `function` соответственно). Эти блоки должны располагаться в тексте программы перед оператором `begin`, начинающим основную программу. В каждом блоке процедуры или функции мо-

жет находиться свой раздел описания переменных (var) и должна быть пара операторов begin … end, между которыми пишется текст процедуры (или функции).

Внимательно читайте комментарии к программе!

Пример 9.1. Демонстрация процедур и функций на примере работы с одномерным массивом

```
Program Massiv_1;
const
 N=10;
type
 Mas=array [1..N] of integer;

var { Раздел описания переменных.
      Все предыдущие разделы описаний, которые мы
      приводили, включая этот, называются глобальными.
      Константы, типы и переменные из этих разделов
      являются глобальными, то есть действуют в теле
      программы, а также во всех процедурах и функциях,
      описанных в этой программе }

   Line: Mas;    { Переменная Line – одномерный массив.
                   Его тип определен нами как Mas }

   Sm  :integer; { Переменная для вычисления
                   суммы элементов }

{ Процедура ввода одномерного массива.
  Внимание! После запуска программы управление
  передается первому исполняемому оператору
  из тела программы! Процедура Inp будет выполняться
  только после вызова ее из тела программы! }

Procedure Inp;   { Заголовок процедуры Inp }

var    { Раздел описания локальных переменных,
         то есть переменных процедуры Inp,
         действующих только в пределах этой процедуры }

   i:integer;    { Переменная, хранящая индекс
                   очередного элемента массива,
                   к которому идет обращение.
```

```
                  Эта переменная i не имеет
                  никакого отношения
                  к глобальной переменной i.
                  Хотя они имеют одинаковое название,
                  это совершенно разные ячейки памяти.
                  В данной процедуре (Inp)
                  обращение к переменной i
                  будет означать локальную переменную.
                  Это, в частности, значит,
                  что к глобальной переменной i
                  из процедуры обратиться нельзя.
                  В данном случае это и не нужно.
                  Если вы хотите иметь возможность
                  обращаться к обеим переменным,
                  нужно давать им разные имена }

begin            { Начало тела процедуры Inp }
   for i:=1 to N do
   begin
      write('Введите элемент с индексом ',i,':');
      readln(Line[i])
   end
end;             { Конец тела процедуры Inp }

{ Процедура вывода одномерного массива на экран }

Procedure Out;   { Заголовок процедуры Out }
var
   i:integer;    { Рекомендуется описывать локально
                   те переменные, которые используются
                   для промежуточных рабочих значений
                   в процедуре, чтобы избежать ошибок
                   в значениях глобальных переменных.
                   Например, рекомендуется локально
                   описывать счетчики циклов }
begin            { Начало тела процедуры Out }
   for i:=1 to N do
      write(Line[i]:5);
   writeln
end;             { Конец тела процедуры Out }

{ Процедура обмена местами соседних элементов }
Procedure Change;
```

```
var
   i:integer;
   X:integer;    { Переменная для временного
                   хранения элементов массива
                   при обмене }

begin            { Начало тела процедуры Change }

   i:=1;         { Начальное значение индекса }
   { При обмене пар элементов индекс изменяется
     с шагом 2. Цикл for ... такой шаг не поддерживает,
     поэтому применяем цикл while ... do }

   while i<10 do { На каждом шаге
                   будем менять i-й и i+1-й элементы }
   begin
      X:=Line[i];         { Первый элемент из пары
                            сохраняем в X }

      Line[i]:=Line[i+1]; { На его место записываем
                            второй элемент из пары }

      Line[i+1]:=X;       { На место второго элемента
                            пишем сохраненный элемент
                            из переменной X }

      i:=i+2              { Увеличение индекса на 2
                            для перехода
                            к следующей паре элементов }
   end
end;             { Конец тела процедуры Change }

{ Функция вычисления суммы элементов массива }

Function Sum:integer;    { Заголовок функции.
                           Через двоеточие указывается
                           тип возвращаемого результата.
                           Результат возвращается
                           через имя функции.
                           В нашем случае возвращаемый
                           результат — сумма элементов
                           массива. Тип результата
                           соответствует типу элементов
                           массива }
```

```
var
   i:integer;
   S:integer;    { Переменная для накопления суммы }

begin            { Начало тела функции Sum }
   S := 0;
   for i := 1 to N do
      S := S + Line[i];

   Sum := S;     { Это обязательная строка!
                   Результат вычислений возвращается
                   в вызывающую программу
                   через имя функции Sum }

end;             { Конец тела функции Sum }

{ Начинается тело программы }

begin            { Именно это и есть главный оператор
                   begin — начало всей программы }

   Inp;      { Вызов процедуры ввода элементов массива }

   Out;      { Вызов процедуры вывода массива на экран }

   Sm := Sum;{ Вызов функции вычисления суммы }

   writeln('Сумма элементов=',Sm);  { Вывод результата
                                      на экран }

   { Результат вычисления функции необходимо присвоить
     переменной того же типа (как в выражении Sm:=Sum)
     или вывести сразу на экран: }
   writeln('Сумма элементов:',Sum);

   Change;   { Вызов процедуры обмена элементов }

   Out;      { Вызов процедуры вывода массива на экран }

   readln
end. { Конец тела программы. Только здесь ставим точку }
```

Задание 9.1. Написать программу, в которой выполняется:

а) ввод одномерного массива A из 14 чисел (положительных и отрицательных);

б) вывод массива на экран;

в) сдвиг всех элементов массива на одну позицию влево (рис. 9.3); первый элемент встает на место последнего (см. блок-схему алгоритма на рис. 9.4);

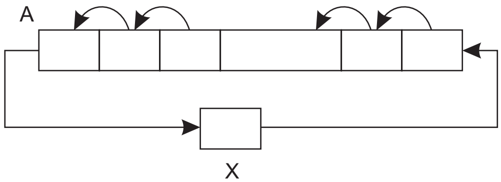


Рис. 9.3. Схема циклического сдвига одномерного массива влево

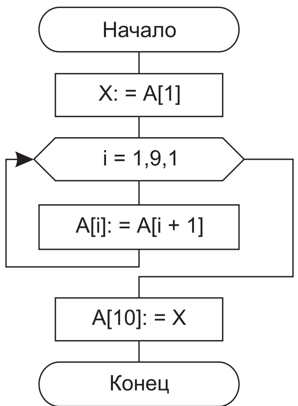


Рис. 9.4. Блок-схема алгоритма циклического сдвига массива влево

г) вывод массива на экран;

д) подсчет количества положительных элементов.

Все пункты оформить как процедуры или функции.

Урок 9.2. Пример работы с функцией: Поиск максимального элемента

Пример 9.2. Программа поиска максимального элемента в массиве

```
Program Maximum;
const
   N=10;
type
   Mas=array [1..N] of integer;
var            { Раздел описания глобальных переменных }

   Line: Mas;  { Переменная Line — одномерный массив.
                 Его тип определен нами, как  Mas }
   m,i : integer;

{ Функция определения индекса
  максимального элемента массива }
function Maxim:integer;

var            { Раздел описания локальных переменных }

   imax:integer;  { Переменная для хранения индекса
                    максимального элемента }

   i   :integer;  { Эта переменная i не совпадает
                    с глобальной переменной i }

begin          { Начало тела процедуры }

   imax:=1;    { За максимальный элемент
                 принимаем первый элемент массива }

   for i := 2 to N do   { Сравним текущего кандидата
                          в максимумы со всеми
                          остальными элементами массива
                          (то есть, начиная со 2-го
                          и до последнего) }

      if Line[imax] < Line[i] then  { Если кандидат в
                                      максимумы оказался
                                      меньше i-го
                                      элемента... }
```

```
          imax:=i;      { ... то считаем i-й элемент
                          новым кандидатом в максимумы }

    Maxim:=imax { Возвращаем найденное значение,
                  присваивая имени функции
                  найденный индекс }

end;            { Конец тела функции }

begin           { Начало тела программы }

    { Заполнение элементов массива
      с использованием датчика случайных чисел }
    Randomize;
    for i := 1 to N do
       Line[i]:=Random(100);

    m:=Maxim;   { Вызов функции поиска индекса
                  максимального элемента массива }

    writeln('Максимальный элемент в массиве=',Line[m]:5);
    writeln('Его индекс=',m:5);
    readln
end.
```

Задание 9.2. Написать программу, в которой для массива из 20 элементов, заполненного случайными числами от –20 до +20, 20 раз выполняется следующая процедура: слева направо по порядку сравниваются все соседние элементы и, если первый элемент в паре оказался больше второго, элементы меняются местами. В конце процедуры массив выводится на экран в строку. Таким образом, результатом программы должны быть 20 строк, в каждой из которых бо́льшие элементы массива постепенно «сдвигаются» вправо, а меньшие — влево.

Выводы

1. При решении сложной задачи разумно ее разбивать на подзадачи. При реализации на языке Паскаль каждая такая подзадача (блок) оформляется в виде функции или процедуры.

2. Процедуры и функции оформляются вне тела программы. Они начинают работать только при вызове из тела программы или из другого блока (функции, процедуры).
3. Результат работы функции возвращается через ее имя.
4. Различают глобальные и локальные переменные.
5. Глобальные переменные действуют в теле программы и в любом из ее блоков (функциях, процедурах).
6. Локальные переменные действуют только внутри блока, в котором они описаны.

Контрольные вопросы

1. Зачем нужно создавать процедуры и функции?
2. Чем отличаются функции от процедур?
3. Зачем нужны локальные переменные?
4. Если локальная и глобальная переменные имеют одинаковые имена, то к какой переменной будет идти обращение?

ТЕМА 10

Как работать с символьными строками

Как известно, основной вид информации, которую хранит, получает и использует человек, — это текстовая информация. В эпоху активного использования вычислительной техники бо́льшая часть информации, обрабатываемой компьютерами, является текстовой. Для удобства ее обработки на компьютере придуманы специальные типы данных и операции над ними.

Урок 10.1. Работаем с цепочками символов: тип String

Для работы с цепочками символов (словами и предложениями) в Turbo Pascal введен специальный тип данных — `String`. Он чем-то похож на массив символов. Однако, в отличие от массива, со строками можно делать больше действий. Например, строки можно складывать.

Описание строковой переменной

Для работы с переменной типа String она должна быть описана в разделе `var`:

```
S : String;
```

В этом случае под строку S выделяется 255 символов, а в памяти, соответственно, она будет представлена 255 байтами. (На самом деле в памяти выделяется 256 байт, но это нам сейчас не важно.)

Если мы не планируем использовать такие большие строки, можно явно указать максимальный размер нужной вам строки. Например, запись `S1 : String[40];` говорит о том, что строка `S1` может содержать от 0 до 40 символов.

Основные действия со строками

Рассмотрим операции, которые можно осуществлять с данными строкового типа (ввод/вывод, присваивание, сравнение).

Пример 10.1. Основные действия с символьными строками

```
Program Line_1;
var
  Name1: String[20]; { Под строку выделено 20 символов }
  Name2: String[20];
  Title: String[40]; { Под строку выделено 40 символов }
  Rez:   String[70];
begin
   Name1:='Д.Прайс_'; { Фактическое число символов
                        в строке должно быть < =
                        числу символов, объявленному
                        в разделе var
                        для этой переменной }

   Title:='Программирование на языке Turbo Pascal';

   Rez:=Name1+Title;    { Строки можно складывать.
                          Это означает, что к
                          символам левой строки
                          будут справа приписаны
                          символы правой строки.
                          Если результат не поместится
                          в строковую переменную,
                          в которую мы его записали,
                          он будет обрезан }

   writeln(Rez);        { На экране имеем:
             Д.Прайс_Программирование на языке Turbo Pascal }

   { К строковым переменным одинаковой длины
     можно применять операции отношения (сравнения): }
   writeln('Введите первое имя:');
   readln(Name1);
   writeln('Введите второе имя:');
   readln(Name2);
   if Name1 = Name2 then
      writeln('Имена одинаковы');
   if Name1 < Name2 then
```

```
      writeln('Первое имя в алфавитном списке раньше');
    readln
  end.
```

ЗАМЕЧАНИЕ

Максимальная длина строки — 255 символов (255 байт). Фактическая длина строки хранится в нулевом байте этой строки (именно поэтому под строку реально выделяется на 1 байт больше).

Задание 10.1. Написать программу, которая выводит в алфавитном порядке три введенных пользователем имени.

Урок 10.2. Некоторые функции и процедуры Паскаля для работы со строками

Для удобства обработки строковых переменных в языке Паскаль предусмотрен ряд процедур и функций. Так как они хранятся в специальном файле-библиотеке подпрограмм, эти подпрограммы принято называть *библиотечными*.

Использование библиотечных подпрограмм работы со строками

В следующем примере мы будем вводить символьную строку, в которой несколько раз встречается слово-образец. Задача состоит в том, чтобы удалить все вхождения этого слова, вставить вместо него слово-заменитель и вывести полученную строку на экран.

Пример 10.2. Поиск и замена

```
Program Line_2;
var
  Str:String;    { По умолчанию для строки
                   выделяется 255 байт }

  Word1,Word2:String[20];   { Переменные для хранения
                              слова-образца
                              и слова-заменителя }

  Len:byte;      { Переменная для хранения
```

```
                    длины слова-образца.
                    Тип byte — для хранения целых чисел
                    в диапазоне 0..255 }

   Position:byte; { Переменная для хранения позиции,
                    с которой начинается в строке
                    слово-образец
                    (его первое вхождение) }
begin
   writeln('Введите слово-образец:');
   readln(Word1);
   writeln('Введите слово-заменитель:');
   readln(Word2);
   writeln('Введите исходную строку:');
   readln(Str);
   Len:=Length(Word1); { Функция Length
                         возвращает длину строки Word1 }
   Position:=Pos(Word1,Str); { Pos(Word1,Str) возвращает
     номер позиции в строке Str, с которой начинается
     первое вхождение слова-образца Word1. Если слово
     в строке отсутствует, то Pos возвращает 0 }

   while Position<>0 do { Пока в строке Str
                          есть слово-образец }
   begin
      Delete(Str,Position,Len); { Процедура Delete
        удаляет Len символов из строки Str, начиная
        с позиции Position, то есть слово-образец }
                                }
      Insert(Word2,Str,Position); { Процедура Insert
        вставляет строку Word2 в строку Str, начиная
        с позиции Position. Таким образом, она
        раздвигает строку }

      Position:=Pos(Word1,Str)  { Вычисляем номер новой
                                  позиции слова-образца
                                  для следующего шага
                                  цикла while }
   end;
  Str:=Copy(Str,2,length(Str)-2); { Функция Copy выдает
        в качестве результата кусок строки такой
        длины, чему равен последний параметр,
        начиная с символа, номер которого указан
```

```
         вторым параметром. В данном случае,
         результатом функции Copy будет строка Str,
         кроме первого и последнего символов }
   writeln(Str);
   readln
end.
```

Задание 10.2. Написать программу, в которой вводится строка из слов с некоторым количеством пробелов между ними. Удалить лишние пробелы, оставив по одному между словами.

Работа с символами строки как с элементами массива символов

В некотором смысле можно считать строковые переменные массивами символов. То есть, тип данных `String` можно рассматривать как `array [1..255] of char`. Это значит, что к элементам строки можно обращаться так же, как к элементам массива — по имени массива (строки) и, в квадратных скобках — номер элемента (номер символа от начала строки). При этом у строк, как вы уже поняли, есть дополнительные операции по сравнению с массивами (их можно складывать, искать в них подстроки, удалять символы и вставлять в них символы). Каким именно способом обращаться к символам строки — строковыми подпрограммами или как к элементам массива — выбирает программист. Например, если в программе требуется добавлять/удалять элементы строки, удобнее пользоваться процедурами `Insert` и `Delete`. А если нужно просмотреть все символы строки — удобнее вспомнить, что строка также является и массивом.

Пример 10.3. Подсчет количества скобок в строке

```
Program String_3;
var
   Str:String;    { Возьмем строку максимальной длины }

   i,k:integer;   { Переменные счетчика цикла и
                    для подсчета числа скобок }
begin
   writeln('Введите строку:');
   readln(Str);

   k := 0; { Изначально число скобок равно нулю }
```

```
    for i:=1 to Length(Str) do { Перебираем по порядку
                                  все символы строки Str }
      { если текущий символ - скобка }
      if (Str[i]='(') or (Str[i]=')') then
        k:=k+1; { Увеличиваем число скобок на единицу }

    writeln(k);
    readln
  end.
```

Выводы

1. Для работы с массивом символов разумнее использовать тип данных String.
2. Со строками можно выполнять операции присваивания, сложения и сравнения.
3. Максимальное количество символов, которое можно хранить в строковой переменной, равно 255.
4. Для удобства работы с типом данных String рекомендуется использовать библиотечные функции и процедуры языка Паскаль.
5. Библиотечные функции позволяют раздвигать строки, вычислять их длину, удалять в них подстроки и осуществлять поиск.

Контрольные вопросы

1. Если в строковой переменной не планируется хранить более 50 символов, как ее разумнее описать в программе?
2. Если к строковой переменной длиной 200 символов, описанной как String, «добавить» (+) строковую переменную длиной 100 символов, какова будет длина получившейся строковой переменной?
3. Как определить длину введенной с клавиатуры строки?
4. Как определить количество точек во введенной с клавиатуры строке?
5. Как увеличить строку вдвое, дописав рядом с каждым символом строки такой же (например, из строки «Вася» получить «ВВаассяя»)?

ТЕМА 11

Процедуры и функции с параметрами

Все вспомогательные алгоритмы, рассмотренные нами в теме 9, были «слепыми». То есть они осуществляли свои действия независимо от каких-либо значений. Однако нетрудно было заметить, что мы пользуемся массой встроенных в Паскаль вспомогательных алгоритмов, действия которых зависят от значений, указываемых нами в скобках. Эти значения называются *параметрами*.

Урок 11.1. Простые примеры использования подпрограмм с параметрами

Уже самый первый оператор, с которым мы познакомились, — `writeln('Привет!')` — является процедурой с параметром. Процедура в данном случае занимается выводом на экран, а параметр указывает, что именно должно появиться на экране.

Наша текущая задача — научиться самим создавать процедуры с параметрами.

Простейшие процедуры с параметрами

Пример 11.1. Использование процедур с параметрами для рисования на экране перекрестий и последовательностей звездочек

```
program Param1;

uses Crt; { Мы будем активно использовать функции
            рисования на экране }

{ Процедура Stars выводит на экран
  указанное количество звездочек }
```

```
procedure Stars(N:integer); { После имени процедуры
                              в скобках указываются
                              ее параметры и их типы.
                              В данном случае процедура
                              имеет один параметр N
                              (типа integer), который
                              указывает количество
                              звездочек, выводимых на
                              экран }

var i:integer; { Раздел описания локальных переменных }

begin
   for i:=1 to N do { Мы используем параметр N
                      для указания того,
                      сколько раз нужно
                      выводить на экран символ "*" }
      write('*')
end;

{ Процедура Cross выводит на экран
  перекрестье с центром в координатах (X,Y) }

procedure Cross(X,Y:integer); { Если параметры
                                однотипные, их имена
                                можно перечислить
                                через запятую }
begin
   gotoxy(x,y-1);    writeln('|');
   gotoxy(x-2,y);  writeln('--+--');
   gotoxy(x,y+1);    writeln('|')
end;

{ Начало основной программы }
begin
   clrscr;        { Очистим экран, чтобы лучше было видно
                    результаты работы }

   Stars(10);     { Вывели 10 звездочек
                    в начало первой строки }

   gotoxy(6,20);
```

```
    Stars(70);     { Заполнили звездочками
                     почти всю 20-ю строку }

    Cross(40,13); { Вывели перекрестье в центр экрана }

    Cross(70,5);  { Вывели перекрестье
                     в правый верхний угол экрана }
    readln
end.
```

Обратите внимание: в процедурах мы используем параметры так, как будто у нас есть переменные с соответствующими именами, и они имеют определенные значения. Это приблизительно так и есть. Эти «переменные» даже можно менять. То есть вполне можно написать `N := N + 2`, и параметр `N` в этом месте действительно увеличится на 2. Нужно только понимать, что «время жизни» этих «переменных» такое же, как у локальных переменных — по окончании работы процедуры они уничтожаются.

Формальные и фактические параметры

Если в качестве параметра при вызове процедуры подставить имя переменной, а внутри процедуры этот параметр изменить, то на саму переменную основной программы это никоим образом не повлияет.

Здесь мы сталкиваемся с понятиями формальных и фактических параметров. Параметры, имена которых используются в процедуре, называются *формальными*. Они могут совпадать или не совпадать по имени с переменными, которые мы подставляем при вызове процедуры.

В момент вызова процедуры значения *фактических* параметров (которые мы подставили в скобки при вызове) копируются в отдельные ячейки памяти, которые используются процедурой и после ее окончания освобождаются. Фактические параметры при этом не изменяются.

Простейшие функции с параметрами

Использование собственных функций позволяет, например, расширить список стандартных функций Паскаля.

Пример 11.2. Применение пользовательских функций с параметрами для расширения набора математических функций языка Паскаль

```
program Param2;

{ Функция вычисления тангенса }
function tg(x:real):real; { Необходимо описать
                            не только тип параметра,
                            но и тип самой функции.
                            В данном случае они
                            совпадают, но, вообще
                            говоря, это не обязательно }
begin
   tg:=sin(x)/cos(x) { В данном случае функция
                       очень простая. Однако ее
                       использование делает программу
                       намного более удобной для чтения}
end;

{ Функция вычисления числа x,
  возведенного в целую положительную степень n }
function Stepen(x:real; n:integer):real;
               { Если параметры разных типов,
                 они перечисляются
                 через точку с запятой }

var
   i:integer;
   y:real; { Аккумулятор для накапливания произведения }
begin
   y:=1;   { Начальное значение
             произведения-аккумулятора
             всегда равно единице }
   for i:=1 to n do
      y:=y*x;
   Stepen:=y
end;
```

```
{ Начало основной программы }
begin
   writeln('Тангенс числа Пи/6=',tg(Pi/6):7:3);

   { Результат функции вещественный, поэтому
     для красивого вывода на экран применяем формат }
   writeln('Число 2.5 в степени 8 равен ',
           Stepen(2.5,8):7:3);

   { Заметьте, использование функций
     делает программу более наглядной }
   writeln('Тангенс Пи/3 в степени 5 равен ',
           Stepen(tg(Pi/3),5):7:3);
   readln
end.
```

Урок 11.2. Способы передачи параметров

Среди стандартных процедур Паскаля можно найти и такие, которые изменяют само значение параметра — например `inc()`. Как самому создать такую процедуру?

Пример 11.3. Процедура, изменяющая значения параметров

```
program Param3;
var a,b:integer;

{ Процедура обмена двух переменных местами }
procedure Change(var x,y:integer);
                  { Для указания того, что значения
                    передаваемых переменных будут изменены
                    в процедуре, используют служебное
                    слово var. Это называется
                    передачей параметров по адресу.
                    Параметр, имя которого используется
                    для передаваемой переменной,
                    называется формальным }

var z:integer; { Временная переменная для обмена }
begin
   z:=x;
   x:=y;
   y:=z
end;
```

```
{ Начало основной программы }
begin
   a:=5;
   b:=8;
   writeln('До обмена A=',a,' B=',b);

   Change(a,b);  { Передаваемая переменная,
                   имя которой указано в скобках
                   при вызове процедуры,
                   называется фактическим параметром.
                   Имена формальных и фактических
                   параметров могут не совпадать }

   writeln('После обмена A=',a,' B=',b);
   readln
end.
```

Названия *формальный параметр* и *фактический параметр* подчеркивают, что при передаче параметров таким образом (со служебным словом var) переменная для формального параметра не создается и память не выделяется. Это имя только формально используется для описания действий, которые будут совершены с данной переменной в подпрограмме. Фактически вместо этого имени используется имя переменной, подставленное при вызове подпрограммы.

Заметим, что при вызове процедуры, изменяющей значения параметров, в качестве фактического параметра нельзя использовать выражения. Фактическим параметром может быть только имя переменной! Это связано с тем, что в такую процедуру передается не значение, а адрес ячейки памяти, в которой хранится переменная-параметр. Если подставить выражение, то Паскаль сообщит об ошибке — ведь выражение не имеет адреса!

Способ передачи параметров, изменяющий их значения, называется *передачей параметров по адресу*. Обычный способ, при котором фактический параметр копируется в отдельную ячейку памяти и который позволяет в качестве параметра передавать значения выражения, называется *передачей параметров по значению*.

Один и тот же вспомогательный алгоритм может получать параметры обоими способами — часть по адресу, остальные по значению.

СОВЕТ

Использование процедур с параметрами, передающимися по адресу, с одной стороны, добавляет дополнительные возможности для программирования, а с другой — часто приводит к ошибкам, которые очень трудно выловить. Изменение значений переменных при вызове процедур и функций не всегда бросается в глаза, особенно, если программа большая и ее написание занимает много дней. Будьте внимательны!

Задание 11.1. Написать процедуру, которая получает целочисленную переменную и «разворачивает» ее цифры в обратном порядке — например число 1234 преобразует в число 4321.

Задание 11.2. Написать процедуру, которая получает две вещественные переменные и возвращает вместо каждой из них отклонение от среднего арифметического. Например, числа 5 и 8 должны превратиться в –1,5 и 1,5, а числа 3,2 и 0,8 — в 1,2 и –1,2.

Выводы

1. Существует возможность создавать свои собственные процедуры и функции с параметрами.
2. Их использование повышает наглядность программы и добавляет ей универсальности.
3. Передаваемые параметры перечисляются в скобках после имени подпрограммы с указанием типа данных.
4. Для изменения значений передаваемых параметров используется передача по адресу.
5. Для указания того, что передача параметров происходит по адресу, используется служебное слово var.
6. Используя передачу параметров по адресу, будьте особенно внимательны!

Контрольные вопросы

1. Приведите пример, когда использование процедуры с параметрами сокращает текст программы втрое.
2. Какой из способов передачи параметров расходует больше памяти?

3. Можно ли передавать в качестве параметра произведение двух переменных при передаче параметров по значению? Где окажется значение произведения? Почему этого нельзя сделать при передаче по адресу?
4. Рассмотрим следующую процедуру:

   ```
   procedure Maxim(var x,y:integer);
   begin
     if x>y then y:=x
     else x:=y
   end;
   ```

   Тело программы выглядит так:

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

   Каковы в результате будут значения переменных x и y?
5. Рассмотрим следующую процедуру:

   ```
   procedure Mult(var x:integer;y:integer);
   begin
     x:=y*2
   end;
   ```

   Тело программы выглядит так:

   ```
   x:=2;
   Mult(x,2*x);
   ```

   Каково в результате будет значение переменной x?

ТЕМА 12

Файлы: сохраняем результаты работы до следующего раза

Как вам должно быть известно из общего курса информатики, память, с которой работает Паскаль и в которой он хранит все свои данные (как и любая другая программа), называется *оперативной*. Она обладает одним неприятным свойством: ее содержимое стирается при выключении питания компьютера. Чтобы информация сохранялась при выключенном питании (принято говорить «для долговременного хранения информации»), используется *внешняя память*. Это разного рода диски, флешки и другие виды носителей. Работе с внешней памятью из программы на Паскале и посвящена наша тема.

Урок 12.1. Как работать с текстовым файлом

Файлом называется порция данных, хранящаяся на диске и имеющая имя. Другими словами, все, что вы пытаетесь сохранить на диске, должно быть записано в виде файла. Для того чтобы работать с файлом в программе, необходимо ввести специальную переменную, которая называется *файловой*. Через нее мы будем записывать и читать информацию из файла.

Основным элементом текстового файла является символьная строка (ASCII). Можно работать как со строкой целиком, так и с каждым символом в отдельности. Обращение к символам, хранящимся в файле, происходит последовательно.

Открытие файла для чтения

Начиная работать с файлом, его *открывают*. При этом в памяти создается особая структура данных, частью которой является

файловый указатель. Это как бы «курсор», который указывает на позицию файла, с которой будет происходить следующая операция чтения (или записи). После чтения символа (или строки) из файла файловый указатель передвигается на следующий символ (строку). При записи в файл эта позиция всегда указывает на конец файла.

Мы начнем с самого простого — попытаемся открыть текстовый файл для чтения и выведем его содержимое на экран. Для того чтобы программе было что открывать, создайте в Блокноте или прямо в среде Turbo Pascal текстовый файл и назовите его `work.txt`. Этот файл должен быть сохранен в той же папке, что и рабочие (`.pas`) файлы с программой на Паскале. Содержимое файла нам не важно. Мы рекомендуем набрать несколько строчек текста, желательно латинскими символами.

Пример 12.1. Вывод на экран содержимого текстового файла work.txt

```
Program File_1;
var
   Fil: text; { Описание файловой переменной
                для работы с текстовым файлом }

   { Паскаль умеет работать с разными типами файлов.
     Мы ограничимся изучением самого распространенного
     и понятного типа — текстового. Такой тип данных
     в Паскале называется text. Удобство работы с
     текстовым файлом состоит в том, что его можно
     открыть в обычном текстовом редакторе
     (например, в Блокноте) и просмотреть или исправить }

   Str : string; { Переменная для чтения
                   строк из файла }
begin

   { Перед началом работы с файлом необходимо выполнить
     несколько предварительных действий: }

   Assign(Fil,'work.txt'); { Файловой переменной Fil
                             назначается конкретное
                             имя файла
                             (в данном случае
                             work.txt) }
```

```
Reset(Fil);          { Файл открывается
                       с признаком "для чтения".
                       Файловый указатель при этом
                       устанавливается
                       в начало файла }

{ Читаем все строки по одной из файла work.txt
  и выводим их на экран. При чтении из файла work.txt
  для определения того, что файл прочитан целиком,
  используется признак "конец файла". Этот признак
  проверяется библиотечной функцией Eof (end of
  file). Пока файловый указатель не достигнет конца
  файла, этот признак, а также результат,
  возвращаемый функцией Eof, имеют значение False
  ("ложь"). Когда конец файла достигнут
  (то есть файл прочитан весь), функция Eof
  возвращает результат True ("истина") }

while not Eof(Fil) do  { Чтение файла происходит
                         до тех пор, пока функция Eof
                         возвращает False. Чтобы условие
                         продолжения цикла выполнялось
                         пока не достигнут конец файла,
                         используется
                         логическое отрицание not }

begin
   readln(Fil,Str);  { Чтение строки
                       из файла work.txt }

   writeln(Str)  { Вывод прочитанной строки
                   на экран }
end;
Close(Fil)       { После окончания работы
                   открытый файл
                   нужно закрыть }
end.
```

Открытие файла для записи

Из предыдущего примера вы, вероятно, поняли, что открытие файла для чтения происходит в результате процедуры `reset`. При этом файловый указатель устанавливается в начало файла, и про-

цедурами чтения из файла `readln` и `read` можно построчно или посимвольно прочитать содержимое всего файла. Если же мы хотим не читать уже имеющийся файл, а записывать в файл свою информацию, мы должны открыть файл для записи. Это делает процедура `rewrite`. Если в текущей папке нет файла с именем, указанным в процедуре `assign`, то создается новый пустой файл. Если файл с таким именем есть, его содержимое очищается. Так или иначе, файловый указатель устанавливается в начало пустого файла, после чего в файл можно записывать информацию построчно (`writeln`) или посимвольно (`write`).

В следующем примере программа создает новый текстовый файл `work.txt` и копирует его содержимое в файл `user.txt`.

Пример 12.2. Создание и копирование файла

```
Program File_2;
var
   Fil_1, Fil_2: text; { Описание файловых переменных
                         для работы с текстовыми файлами }

   Str : string;   { Переменная для чтения
                     строк из файла и записи в него }

{ Процедура создания нового файла }
Procedure New_file;
var   I : integer;
begin
   Assign(Fil_1,'work.txt');
   Rewrite(Fil_1);      { Создается новый пустой файл
                          и открывается с признаком "для
                          записи". Файловый указатель
                          устанавливается в начало файла
                          (которое в данном случае
                          совпадает с концом).
                          Файл пуст }

   { Введем с клавиатуры три любых символьных строки
     и запишем их в файл work.txt }
   for I:=1 to 3 do
   begin
      writeln('Введите строку и нажмите Enter:');

      readln(Str);       { Введенная строка помещается
                           в переменную Str }
```

```
      writeln(Fil_1,Str) { Строка Str записывается
                           в файл. Файловый указатель
                           перемещается в начало
                           следующей строки, то есть
                           в то место, куда будет
                           внесена следующая запись }
   end;
   Close(Fil_1)          { Файл закрывается.
                           Это обязательно нужно делать
                           после окончания работы
                           с файлом или перед его новым
                           открытием }
end;
{ Чтение всех строк по одной из файла work.txt
  и запись их в файл user.txt }
Procedure Copy_file;
begin
   Assign(Fil_1,'work.txt');

   Reset(Fil_1);         { Файл открывается
                           с признаком "для чтения".
                           Файловый указатель при этом
                           устанавливается
                           в начало файла }

   Assign(Fil_2,'user.txt'); { Файловой переменной Fil_2
                               назначается имя файла,
                               в который будет
                               идти копирование }

   Rewrite(Fil_2);       { Создается новый файл
                           и открывается "для записи". }

   while not Eof(Fil_1) do  { Чтение файла происходит
                              до тех пор, пока функция Eof
                              возвращает False. Чтобы условие
                              продолжения цикла выполнялось
                              пока не достигнут конец файла,
                              используется
                              логическое отрицание not }
   begin
      readln(Fil_1,Str);  { Чтение строки
                            из файла work.txt }
```

```
          writeln(Fil_2,Str)  { Запись прочитанной строки
                                в файл user.txt }

       end;
       Close(Fil_1);
       Close(Fil_2)           { После окончания работы
                                все открытые файлы
                                положено закрывать }
    end;
    begin                     { Начало основной программы }

       New_File;  { Вызов процедуры создания нового файла }

       Copy_file  { Вызов процедуры копирования в файл }

    end.
```

После работы программы откройте оба файла (это можно сделать в режиме Open в среде Turbo Pascal или в Блокноте) и убедитесь, что все получилось верно.

ЗАМЕЧАНИЕ

Если вы вводили эти три строки по-русски, то в Блокноте вы, скорее всего, увидите странную путаницу из русских букв. Это оттого, что Turbo Pascal — программа для MS-DOS, и символы, которые вы вводили, тоже были записаны в кодировке MS-DOS. Чтобы их нормально прочитать, нужно или открывать эти файлы из программы для MS-DOS (например, Norton Commander), или использовать преобразование формата (это умеет делать, например MS Word).

Задание 12.1. Напишите программу, которая:

а) создает текстовый файл из четырех строк строчных латинских букв;

б) читает строки из созданного файла и преобразовывает их в строки заглавных латинских букв;

в) после преобразования каждую строку записывает в другой созданный текстовый файл.

Проверьте результаты работы путем чтения обоих файлов!

Урок 12.2. Сохранение двумерного массива чисел в текстовом файле

Числовые данные тоже можно сохранять в текстовых файлах. Так как текстовый файл можно будет читать не только программно, но и средствами обычного редактора, это очень удобно: можно сохранить в файле числовые результаты и вставить их потом в другую программу.

Сохранение числовых данных в текстовом файле

Пример 12.3. Сохранение чисел в текстовом файле

```
Program File_3;
var
   A,B:integer;
   Fil:text;
begin
   A:=3;
   B:=10;
   Assign(Fil,'prim.txt');
   Rewrite(Fil);

   write(Fil,A,' ');    { Файловый указатель в файле
                          prim.txt после этой операции
                          будет стоять за пробелом
                          после числа 3, без перехода
                          на следующую строку }

   write(Fil,B);        { При записи числа
                          в текстовый файл записывается
                          его символьный код }

   Close(Fil)
end.
```

Задание 12.2. Просмотреть полученный файл prim.txt в текстовом редакторе.

Сохранение массива чисел в текстовом файле

Рассмотрим, как можно записывать в текстовый файл числовые массивы. Это очень удобный способ обмена информацией: можно подготовить данные на одном компьютере, сохранить их в файл, перенести его на другой компьютер и там открыть в другой программе.

Пример 12.4. Запись матрицы вещественных чисел 5×4 в текстовый файл и чтение данных из файла

```
program File_4;
const
   M=5; N=4;
var
   Fil:text;
   A:real;
   S:char;
   I,J: integer;
begin
   assign(Fil,'matrix.txt');
   rewrite(Fil);
   randomize;

   { Запись матрицы в текстовый файл }
   for I:=1 to M do
   begin
      for J:=1 to N do
      begin
         A:=random(100);      { Запишем в элемент
                                матрицы случайное
                                число }

         write(Fil,A:5:3,' ') { Число A записывается
                                в файл в указанном
                                формате, за ним стоит
                                пробел }

      end;

      writeln(Fil)            { i-я строка массива
                                кончилась.
```

```
                              Файловый указатель
                              переводится на следующую
                              строку в файле }

end;
close(Fil);
{ Чтение файла и вывод матрицы на экран по строкам.
  Повторное использование процедуры assign(Fil)
  не требуется }
reset(Fil);

while not eof(Fil) do
begin
   while not eoln(Fil) do  { eoln(Fil) возвращает
                             статус конца строки.
                             Возвращаемый результат
                             имеет тип boolean.
                             Пока не будет достигнут
                             конец строки, цикл
                             выполняется.
                             Функция eoln работает
                             только для текстового
                             файла (для него имеет
                             смысл понятие строки) }
   begin
      read(Fil,A);         { Чтение числа из текущей
                             строки файла }

      write(A:5:3);        { Вывод числа на экран }

      read(Fil,S);         { Чтение пробела
                             за числом.
                             Пробел — это символ.
                             Он не может быть
                             считан как число
                             в переменную A }

      write(S)             { Вывод пробела на экран }
   end;
   writeln;                { Перевод курсора
                             на экране
                             на следующую строку }
```

```
      readln(Fil)              { Перевод
                                 файлового указателя
                                 на следующую строку }
    end;
    close(Fil);
    readln
  end.
```

Пример 12.5. Чтение данных из текстового файла в двумерный массив

```
  program ReadMatrix;
  const
    M=5; N=4;
  var
    Fil:text;
    A:array[1..M,1..N] of real;
    S:char;
    I,J: integer;
  begin
    assign(Fil,'matrix.txt');

    reset(Fil);     { Открываем для чтения тот же файл,
                      в который записали матрицу }

    { Чтение матрицы из текстового файла }
    for I:=1 to M do
    begin
      for J:=1 to N do
      begin

        read(Fil,A[I,J],S);   { Читаем из файла
                                элементы построчно.
                                В переменную S
                                считываем пробел
                                между числами }

        write(A[I,J]:6:3,' ') { Выводим матрицу
                                на экран
                                в том же порядке }
      end;

      writeln;                { Переходим на следующую
                                строку на экране }
```

```
      readln(Fil)              { Переходим на следующую
                                 строку файла }
   end;
   closefile(Fil);
   readln
end.
```

Задание 12.3. Написать программу чтения файла `prim.txt` и вывода чисел из файла на экран монитора.

Задание 12.4. Используя файл `matrix.txt`, подсчитать сумму элементов в каждой строке и записать полученные результаты в новый файл `rezult.txt`.

Дописывание информации в конец файла

Процедура `rewrite`, как уже было сказано, очищает все содержимое файла, если он уже существует. Если же нужно дописать что-то к уже существующему файлу, используется процедура `append`. Она тоже открывает файл для чтения, но не очищает старое содержимое файла, а устанавливает файловый указатель в конец.

Пример 12.6. Дописывание информации в конец текстового файла

```
program File_6;
const
   M=5; N=4;
var
   Fil:text;
   append(Fil);                { Открытие файла
                                 для добавления текста
                                 в конец }

   writeln(Fil,'Конец матрицы'); { Допишем этот текст
                                   в конец записанной матрицы }

   close(Fil);
   readln
end.
```

ЗАМЕЧАНИЕ

Вы, конечно, обратили внимание, что процедуры вывода на экран (`write` *и* `writeln`) *имеют те же названия, что и процедуры записи в файл.*

То же касается и процедур `read` *и* `readln`. *Это оттого, что экран и клавиатура с точки зрения Паскаля также являются файлами. Эти файлы называются* `con` *(консоль). То есть, когда мы пользуемся процедурой вывода на экран, Turbo Pascal на самом деле выводит информацию в файл* `con`, *под которым система понимает монитор. А когда мы читаем информацию с клавиатуры, Turbo Pascal читает информацию из файла* `con`, *под которым система понимает клавиатуру. Вы можете убедиться в этом сами: присвойте файловой переменной имя файла* `con (assign(Fil,'con'))` *и попробуйте открыть этот файл для записи* (`rewrite(Fil)`) *и записать в него какие-нибудь данные (например,* `writeln('Hello, world!')`*). Этим же методом можно выводить из Turbo Pascal информацию на принтер. Нужно только знать, что имя файла-принтера —* `lpt1`.

Выводы

1. Для долговременного хранения полученной информации используются файлы.
2. Нами рассмотрена работа с текстовыми файлами.
3. Основным элементом файла типа `text` является строка символов ASCII.
4. Для работы с файлом вводится файловая переменная, через которую идет обращение к файлу.
5. При работе с файлом необходимо учитывать положение файлового указателя, который перемещается к концу файла по мере чтения.
6. Для работы с файлом необходимо назначить файловой переменной имя файла с помощью команды `assign` и открыть его для чтения (`reset`) или для записи (`rewrite`).
7. Запись в файл и чтение из файла производятся процедурами `write`/`writeln` и `read`/`readln` с обязательным указанием в качестве первого параметра имени файловой переменной.
8. После окончания работы с файлом его необходимо закрыть командой `close`.

Контрольные вопросы

1. В какой памяти хранятся данные, которые обрабатывает программа? В какой памяти нужно сохранять эти данные, если они должны храниться долго?
2. На что указывает файловый указатель?
3. Какие действия необходимо совершить, чтобы открыть файл для чтения?
4. На диске хранился файл `example.txt`, в котором находилось 5 страниц текста (примерно 9 Кбайт). Программа выполнила следующие действия:

   ```
   assign(f,'example.txt');
   rewrite(f); writeln(f,'Конец');
   close(f);
   ```

   Как изменится количество информации, содержащейся в файле?
5. Ответьте на предыдущий вопрос, если вместо операции `rewrite` поставить операцию `append`.
6. В файле хранится строка из 10 цифр (от 0 до 9), разделенных пробелом. Программа считывает их из файла следующим образом:

   ```
   for i:=1 to 9 do
   begin
     read(f,c); x:=ord(c)
   end
   ```

   Чему будет в результате равна переменная `x` (`i,x:integer; c:char;`)?

ТЕМА 13

Графический режим работы. Модуль Graph

Мы выводили на экран узоры из звездочек, но, безусловно, это изображение не являлось графическим. Такие изображения из символов иногда называют псевдографикой. Монитор работал в текстовом режиме — 80×25 символов на экране (см. урок 1.2).

В этой теме мы познакомимся с возможностями среды Turbo Pascal для работы с графической информацией.

Урок 13.1. Включаем графический режим работы

Все предыдущие задачи красивого вывода на экран мы решали, используя текстовый режим работы и библиотеку Crt, обслуживающую этот режим.

Особенности работы с графикой

При работе в графическом режиме изображение на экране строится не из символов, а из точек — пикселов. Каждый пиксел имеет две координаты, x и y (рис. 13.1), и определенный цвет (по умолчанию белый).

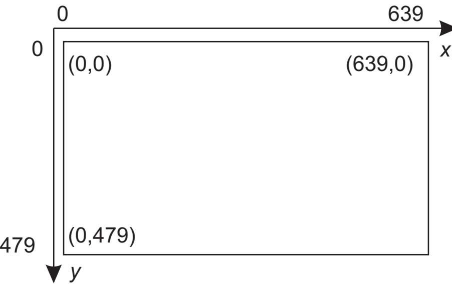


Рис. 13.1. Система координат пикселов в графическом режиме работы

При использовании модуля Graph, обслуживающего графический режим, Turbo Pascal умеет работать с разрешениями экрана до 640×480 пикселов.

ВНИМАНИЕ

Для работы программы в графическом режиме необходима специальная программа — драйвер графического режима. В обычной установке среды Turbo Pascal такой драйвер хранится в файле `egavga.bgi`. *Рекомендуется скопировать его в свой текущий каталог.*

Местоположение этого файла можно найти в режиме Поиск или поискать в каталоге BGI установки Turbo Pascal.

Левый верхний пиксел имеет координаты (0,0). Количество пикселов зависит от типа дисплейного адаптера и режима его работы. Для современных компьютеров это разрешение (640×480) считается уже устаревшим. Но для работы с более высоким разрешением требуется современный драйвер экрана (не `egavga.bgi`, а `svga.bgi`, например). Он не входит в стандартную поставку Turbo Pascal, поэтому мы не будем его рассматривать.

Переключение в графический режим видеоадаптера

Стандартное состояние компьютера при запуске среды Turbo Pascal соответствует работе экрана в текстовом режиме. Поэтому для использования графических средств надо инициировать графический режим работы дисплейного адаптера, то есть переключить экран в графический видеорежим. Для этого подключается графический драйвер — специальная программа, осуществляющая управление теми или иными техническими средствами (монитором и видеоадаптером).

Все стандартные процедуры и функции для работы в графическом режиме содержатся в библиотечном модуле `Graph`. Поэтому необходимо подключить его в разделе объявления дополнительных модулей.

Пример 13.1. Заполнение экрана разноцветными точками

```
uses
   Graph, Crt;    { Подключаем модули.
                    Crt нам также понадобится }
```

```
var
   Gd,Gm:Integer; { Переменная Gd определяет
                    тип драйвера адаптера }

   { Переменная Gm определяет режим работы адаптера;
     по умолчанию выбирается старший режим
     (с самым высоким разрешением) }

   Color : byte;

begin
   Gd := Detect;  { Тип драйвера адаптера
                    определяем автоматически }

   InitGraph(Gd,Gm, ‘’); { Инициализация графики.
                           В кавычках указывается путь
                           к программе-драйверу
                           с расширением bgi.
                           Сейчас предполагается,
                           что драйвер находится
                           в вашем текущем каталоге
                           (откуда вы запускали
                           Turbo Pascal).
                           Если при запуске возникнет
                           ошибка, проверьте,
                           где на самом деле
                           хранится файл egavga.bgi,
                           и укажите этот путь
                           в качестве третьего параметра
                           InitGraph. Например,
                   InitGraph(Gd,Gm,’c:\sys\tp71\bgi’) }

   If GraphResult <> grOK then { Если инициализация
                                 не была успешной –
                                 остановка }

      Halt(1);
   Randomize;

   { На экран выводятся разноцветные точки
     внутри квадрата 100x100,
     пока вы не нажмете любую клавишу }
   repeat
      Color := Random(15);
```

```
      PutPixel(Random(100),Random(100),Color);
      { Процедура PutPixel(X,Y,C) перекрашивает пиксел
        с координатами (X,Y) в цвет C.
        Мы задаем координаты случайным образом }

      Delay(10)
    until KeyPressed;

    CloseGraph      { Эта процедура выключает
                      графический режим
                      и снова делает экран
                      текстовым }

end.
```

Задание 13.1. Измените программу так, чтобы:

- пикселы светились по всему экрану (640×480);
- пикселы светились в верхней половине экрана;
- пикселы светились в нижней половине экрана (добавляйте постоянное приращение к случайной координате).

Урок 13.2. Продолжаем изучать возможности модуля Graph

Мы рассмотрели, как переключиться в графический режим работы и как выводить на экран точки определенного цвета. Но одними точками возможности модуля Graph не ограничиваются. Он умеет также рисовать простейшие геометрические фигуры — линии, прямоугольники и окружности.

Рисование линий средствами модуля Graph

Рассмотрим пример рисования линий. Цвет рисуемой линии устанавливается процедурой `SetColor`. Сами линии рисуются процедурой `Line`.

Пример 13.2. Вычерчивание линий в цикле

```
uses
   Graph, Crt;
var
```

```
   Gd,Gm:Integer;
   Color : byte;
begin
   Gd := Detect;
   InitGraph(Gd,Gm,'');
   If GraphResult <> grOK then
      Halt(1);
   Randomize;

   { На экран выводятся разноцветные линии (отрезки)
     внутри квадрата 200x200,
     пока вы не нажмете любую клавишу }

   { Процедура Line(X1,Y1,X2,Y2) рисует отрезок.
     Начало отрезка — координаты (X1,Y1),
     конец отрезка — координаты (X2,Y2).
     Координаты мы задаем случайным образом.
     Процедура SetColor(цвет) устанавливает цвет линий.
     Цвет мы тоже задаем случайно! }

   repeat
      SetColor(Random(15));
      Line(Random(200),Random(200),
           Random(200),Random(200));
      Delay(10)
   until KeyPressed;
   Readln;
   CloseGraph

   { Эта процедура выключает графический режим
     и снова переводит экран в текстовый вид.
     Именно поэтому важно было написать readln
     перед CloseGraph, иначе мы бы не увидели
     результата рисования (при переходе
     в текстовый режим вся картинка графического режима
     пропадает) }

end.
```

Задание 13.2. Измените программу так, чтобы линии выводились по всему экрану (640×480), все отрезки начинались бы в центре, а конец отрезков задавался бы случайно (рис. 13.2).

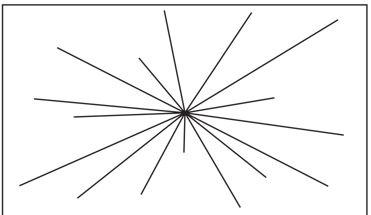


Рис. 13.2. Пояснительный рисунок к заданию 13.2

Задание 13.3. Измените программу так, чтобы на экране было два пучка отрезков: из левого нижнего угла и из правого верхнего угла (рис. 13.3).

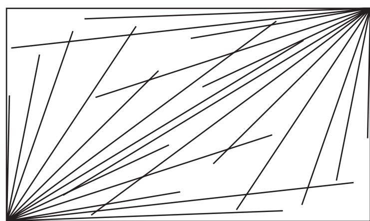


Рис. 13.3. Пояснительный рисунок к заданию 13.3

Рисование окружностей средствами модуля Graph

Теперь рассмотрим, как в графическом режиме можно рисовать окружности. Этим занимается процедура `Circle`.

Пример 13.3. Вычерчивание разноцветных концентрических окружностей

```
uses
   Graph, Crt;
var
   Gd,Gm,r:Integer;
   Color : byte;
begin
   Gd := Detect;
   InitGraph(Gd,Gm,'');
   If GraphResult <> grOK then
```

```
    Halt(1);
  Randomize;

  { На экран выводится 10 разноцветных окружностей }
  for r:=1 to 10 do
  begin
    SetColor(random(16));

    Circle(320,240,r*5) { Процедура Circle(X,Y,R)
                          рисует окружность
                          с центром в точке (X,Y)
                          и радиусом R }

  end;
  Readln;
  CloseGraph
end.
```

В модуле Graph имеется еще очень много процедур (например, `ellipse, bar, rectangle, outtext`), однако их мы предлагаем вам изучить самостоятельно.

СОВЕТ

Если вы наберете в редакторе Turbo Pascal слово Graph, установите на него курсор и нажмете Ctrl+F1*, то вы получите справку по процедурам и функциям модуля* `Graph`*. Постарайтесь добраться до готовых примеров, скопировать их в свою программу и поэкспериментировать.*

Выводы

1. Работать с монитором можно в графическом режиме. Для этого надо в своем рабочем каталоге иметь файл с драйвером — в нашем случае `egavga.bgi`.
2. При работе в графическом режиме изображение строится из пикселов (640×480). Их количество (иначе говоря, разрешающая способность экрана) определяется типом адаптера монитора и его режимом работы.
3. Процедуры для работы в графическом режиме хранятся в библиотечном модуле Graph.

4. Для переключения в графический режим работы используется процедура `InitGraph`. Для возврата в текстовый режим — `CloseGraph`.
5. Библиотечный модуль Graph имеет очень много процедур для рисования различных геометрических фигур. Для получения справки по процедурам модуля Graph нужно установить курсор на слово Graph в программе и нажать `Ctrl+F1`.

Контрольные вопросы

1. Чем отличаются друг от друга графический и текстовый режимы работы?
2. Из каких элементов строится изображение в графическом режиме работы?
3. Какие координаты имеет точка, находящаяся в правом нижнем углу экрана? А в центре экрана?
4. Какие команды нужно написать, чтобы нарисовать две перекрещивающиеся линии, идущие по диагоналям экрана?
5. Какие команды нарисуют ряд концентрических окружностей, расходящихся из центра экрана на равном расстоянии друг от друга?

ТЕМА 14

Операторы, изменяющие естественный ход программы

Язык Паскаль задумывался как структурный язык. То есть любой алгоритм в нем можно описать в виде набора операторов условия и цикла, каждый из которых можно рассматривать как отдельный блок. В блоки «вкладываются» более мелкие блоки, и т. д. Поэтому реализация программы легко осуществляется в рамках структурного программирования.

Ряд языков программирования (таких, например, как Фортран и Бейсик) не удовлетворяют свойству структурности: в них для описания алгоритма приходится использовать, например, оператор безусловного перехода (`goto`).

Для первых языков программирования использование безусловного перехода являлось совершенно естественным, так как сама инструкция безусловного перехода используется в каждой программе, написанной на машинном коде. Без нее нельзя реализовать такие, например, конструкции, как `if...else`.

Первоначально языки программирования придумывались как средство более удобной записи машинных команд, поэтому в них оператор `goto` применялся очень широко.

Однако после была выработана идея, что использование в программе безусловного перехода сильно запутывает программу, особенно если этот переход осуществляется «наверх», то есть возвращает нас в программе к тем операторам, которые уже выполнялись. Тезис структурного программирования призывает вовсе не использовать в программе оператор `goto`. Для людей, которые, вероятно, первый раз о таком слышат, это кажется вполне естественным. А вот для программистов, которые привыкли мыслить «в терминах `goto`», отказ от его использования был очень странен и вызывал много возражений.

Поэтому для облегчения перехода на Паскаль программистов старой школы и для тех редких случаев, когда использование goto оказывается более удобным, оператор был оставлен в языке.

В этой теме мы рассмотрим три оператора, изменяющих обычный ход программы, без использования которых вполне можно обойтись. Рассказываем мы о них для полноты изложения и из-за удобства их применения в ряде случаев.

Урок 14.1. Использование оператора безусловного перехода goto

Споры между противниками и сторонниками goto не утихают до сих пор. Мы приведем один из основных примеров оправданного использования goto.

Рассмотрим задачу проверки элементов массива на уникальность (определить, все ли элементы массива различны).

Проанализируем задание и методы его решения.

Что означает, что все элементы различны? Это значит, что в массиве нет ни одной одинаковой пары элементов.

А как это проверить? Задачу проще решать от противного: постараемся найти в массиве два одинаковых элемента. Если таких не найдется, значит, все элементы различны.

Так как одинаковые элементы могут быть как угодно разбросаны по массиву, необходимо сравнить каждый элемент с каждым. То есть нужно сравнить первый элемент со всеми остальными (это цикл), затем второй элемент со всеми остальными (и это цикл), и так перебрать все элементы. Это означает, что мы должны использовать вложенные циклы. Всего при этом у нас получится около N^2 сравнений (точнее, $(N^2-N)/2$).

Заметим, что если мы в какой-то момент найдем совпадающую пару элементов, перебирать все оставшиеся будет уже не обязательно. Значит, нужно выйти из обоих циклов. Вот для этого нам и понадобится оператор goto.

Пример 14.1. Проверка элементов массива на уникальность

```
program use_goto;

label konec;  { Чтобы указать, в какое место программы
                будет переход оператором goto,
```

```
                это место необходимо пометить.
                В разделе Label описывается имя будущей
                метки, чтобы Паскаль не посчитал метку
                ошибочным оператором }

const N=10;     { Размер массива }

var a:array[1..N] of integer;

   i,j:integer; { Счетчики циклов }

   fl:boolean;
begin
   { Заполним массив и выведем его на экран }
   randomize;
   for i:=1 to N do
   begin
      a[i]:=random(15);
      write(a[i]:4)
   end;
   writeln;

   fl:=false;  { Переменная fl (флаг) нужна нам
                 для того, чтобы по выходе из цикла
                 определить, вышли мы по окончании
                 обоих циклов или раньше }

   for i:=1 to N-1 do
      for j:=i+1 to N do
         if a[i] = a[j] then
         begin
            fl:=true;

            goto konec   { переходим к метке "konec" }

         end;

   konec:      { Это место программы мы и пометили
                 как цель безусловного перехода }

   if fl then
      writeln('В массиве есть одинаковые элементы')
   else
```

```
      writeln('Все элементы массива уникальны');
   readln
end.
```

Итак, использование goto считается оправданным, если таким способом происходит выход из нескольких вложенных циклов вперед.

Справедливости ради приведем пример той же программы без использования goto.

Пример 14.2. Проверка элементов массива на уникальность без использования goto

```
program without_goto;

const N=10;      { Размер массива }

var a:array[1..N] of integer;

   i,j:integer;  { Счетчики циклов }

   fl:boolean;
begin
   { Заполним массив и выведем его на экран }
   randomize;
   for i:=1 to N do
   begin
      a[i]:=random(15);
      write(a[i]:4)
   end;
   writeln;

   fl:=false;    { Переменная fl (флаг) нужна нам
                   для того, чтобы по выходе из цикла
                   определить, вышли мы по окончании
                   обоих циклов или раньше }

   { Вместо for приходится использовать while,
     так как появляется еще одно условие окончания цикла
   }
   i:=1;
   while (j<N) and not fl do  { В каждом цикле мы
                                проверяем два условия:
                                выход за границы массива
                                и флажок окончания
                                поиска }
```

```
    begin
      j:=i+1;
      while (j<=N) and not fl do
      begin
        if a[i] = a[j] then
          fl:=true;
        j:=j+1
      end;
      i:=i+1
    end;
    if fl then
      writeln('В массиве есть одинаковые элементы')
    else
      writeln('Все элементы массива уникальны');
    readln
  end.
```

Программа стала чуть сложнее из-за применения while вместо for, зато мы обошлись без goto.

Урок 14.2. Операторы, изменяющие ход выполнения цикла

В Паскале имеются еще два оператора, действие которых напоминает действие оператора безусловного перехода. Оба они применяются для изменения хода выполнения цикла (напоминаем, что в Паскале есть три вида циклов — for, while и repeat).

Оператор break

Оператор break прерывает действие текущего цикла и передает управление тому оператору программы, который должен выполниться после окончания цикла.

Разумеется, оператор break должен находиться внутри тела цикла.

Рассмотрим применение оператора break.

Пример 14.3. Вывод на экран первого по счету двузначного числа, сумма квадратов цифр которого делится на 17

```
var i:integer;
begin
  for i:=10 to 99 do
    if (sqr(i mod 10) + sqr(i div 10)) mod 17 = 0
```

```
      then break;
   writeln(i);
   readln
end.
```

Вместо применения оператора break можно использовать переменную-флажок, по значению которой определять, нужно ли выполнять цикл далее.

В данном случае (пример 14.3), так как факт наличия нужного числа считался заранее известным, проще было бы использовать цикл while, проверяющий только искомое условие.

Пример 14.4. Вывод на экран первого по счету двузначного числа, сумма квадратов цифр которого делится на 17 (без оператора break)

```
var i:integer;
begin
   i:=10;
   while (sqr(i mod 10) + sqr(i div 10)) mod 17 <> 0 do
      i:=i+1;
   writeln(i);
   readln
end.
```

Оператор continue

Оператор continue заканчивает выполнение текущего шага цикла. То есть после оператора continue сразу выполнится проверка условия окончания цикла. Если цикл еще должен продолжаться, начнет снова выполняться тело цикла.

Оператор continue, так же как и break, должен находиться внутри тела цикла.

Рассмотрим применение оператора continue.

Пример 14.5. Вывод на экран всех двузначных чисел, сумма квадратов цифр которых не делится на 17

```
var i:integer;
begin
   for i:=10 to 99 do
   begin
      if (sqr(i mod 10) + sqr(i div 10)) mod 17 = 0
      then continue;
      writeln(i)
```

```
  end;
  readln
end.
```

Как и в случае с break, использование оператора continue не является обязательным.

Выводы

1. Паскаль — структурный язык программирования. Любая программа на нем может быть составлена из блоков.
2. Для удобства работы программистов старой школы в языке сохранен оператор безусловного перехода goto. Он позволяет выйти из блока операторов в другую точку программы.
3. Для указания той точки, куда может выйти оператор goto, используется метка — произвольный идентификатор, после которого стоит двоеточие.
4. Идентификатор метки должен быть описан в разделе label.
5. На одну метку может ссылаться несколько операторов goto.
6. Для досрочного завершения выполнения тела цикла можно использовать операторы break и continue.
7. Оператор break прекращает выполнение цикла. После его выполнения программа выполняет оператор, следующий после цикла.
8. Оператор continue прекращает выполнение текущего тела цикла. После его выполнения программа проверяет условие окончания цикла.

Контрольные вопросы

1. Какую функцию выполняет оператор безусловного перехода goto?
2. Почему нежелательно использовать в программе оператор goto?
3. Чем отличается действие оператора break от действия оператора continue?
4. В каком случае в программе удобно использовать оператор break? Можно ли без него обойтись?

ТЕМА 15
Группируем данные: записи

Этой темой мы начинаем вторую часть нашей книги. В ней обсуждаются удобные для некоторых применений структуры данных. Для того чтобы программировать, излагаемое здесь и далее вовсе не является обязательным. Но, как вы прекрасно понимаете, мы не стали бы этого писать, если бы это было лишним. Применение этих методов позволяет сделать программы более удобными, наглядными и эффективными. А ведь сам Вирт (создатель языка Паскаль) отмечал, что правильно выбранная структура данных — половина успеха программы.

В этой главе обсуждается способ структурировать, группировать разнотипные данные, которые описывают тот объект, с которым работает ваша программа. Заметим, что описываемый в этой главе тип данных — первый шаг к объектно-ориентированному программированию.

Предположим, вы собираетесь хранить в программе информацию о некоем ученике. Для этого вам нужно хранить его фамилию, имя, рост, вес, пол, номер класса. Все эти параметры имеют разный тип данных (фамилия, имя и класс — строковые, рост — целый, вес — вещественный, пол экономнее хранить как `boolean`). Можно, конечно, выделить для каждого параметра отдельную переменную. Но тогда эти переменные не будут связаны между собой. Нам самим придется помнить, что этот вот набор переменных соответствует этому объекту, а вот тот — другому! Гораздо удобнее было бы иметь некий способ объединить все эти переменные в единую конструкцию! Особенно если таких объектов (учеников) в вашей программе планируется несколько.

Вот именно о таком способе объединения данных и пойдет речь далее.

Урок 15.1. Описание типа данных record

В языке Паскаль и в базах данных используется одинаковая терминология. В базах данных информация об одном объекте называется *записью*, а параметры этого объекта — *полями*. В Паскале так же. Причем тип данных, который позволяет это сделать, так и называется — *запись* (record).

Описание структуры данных типа запись:

```
record <перечисление названий и типов данных полей через точку
с запятой> end;
```

Пример 15.1. Описание переменных типа Запись

```
var
   Vasya,Petya: record
                   Name,Family,Class:string[20];
                   Height:integer;
                   Weight:real;
                   Men:boolean
                end;
```

Как видите, между словами record и end описаны как бы обычные переменные с указанием их типов данных. Однако использование этих переменных оказывается чуть сложнее и при этом гораздо удобнее.

ЗАМЕЧАНИЕ

Нам бы хотелось лишний раз обратить ваше внимание на важность выбора имен переменных и типов данных. Действительность такова, что программы чаще читают, чем пишут. В частности, это связано с тем, что большие программы разрабатывают не в одиночку, а группами (иногда очень большими). Естественно, при этом разные люди изучают одну и ту же программу на предмет исправления в ней ошибок или внесения в нее дополнительных функций. Поэтому особенно важно, чтобы ваша программа была сразу понятна любому человеку, который будет ее читать. Для этого нужно выбирать адекватные и емкие по смыслу имена переменных, а также всегда писать комментарии к программе. В частности, описывать назначение используемых переменных.

Пример 15.2. Использование переменных типа Запись

```
var
   Vasya,Petya: { Обрабатываем данные о двух учениках }
    record
      Name,              { Имя ученика }
      Family,            { Фамилия ученика }
      Class:string[20]; { Название его класса }
      Height:integer;    { Его рост }
      Weight:real;       { Его вес }
      Men:boolean        { Его пол (мужской? да/нет) }
    end;
begin
  Vasya.Name:='Василий'; { Для обращения к конкретному
                          полю переменной типа запись
                          нужно написать имя переменной,
                          точку и имя поля }
{ Работать с такой конструкцией можно как с обычной
  переменной }
  readln(Vasya.Family);
  Vasya.Height:=170;
  Vasya.Height:=Vasya.Height+5;
  Vasya.Weight:=65.3;
  Vasya.Men:=true;
  Vasya.Class:='10a';
  Petya:=Vasya; { Однотипные переменные типа Запись
                  можно присваивать одну другой.
                  При этом копируется содержимое
                  всех полей записи }
  Petya.Name:='Петр'
end.
```

Термин *однотипные* в этом примере означает, что переменные либо должны быть описаны в разделе `var` через запятую, либо необходимо объявлять в разделе `type` свой собственный тип данных с определенным именем и тогда эти переменные должны быть описаны как такой тип данных. Если же для каждой из них дать отдельное описание (свою конструкцию `record ... end`), пусть даже они будут полностью идентичны, Паскаль будет считать их принадлежащими к разным типам данных.

ЗАМЕЧАНИЕ

Использование в программе записей — еще один случай (кроме оператора `case`*), когда количество операторов* `end` *в программе не должно совпадать с количеством операторов* `begin`*.*

Урок 15.2. Когда и как разумно использовать записи

Использование записей не является обязательным. Они просто облегчают работу с данными, которые вы обрабатываете — структурируют их. Записи имеет смысл использовать всегда, когда в вашей программе обрабатываются объекты, описываемые несколькими параметрами.

Например: координаты точки на плоскости (`record x,y:real end`); фамилия, имя и пол человека (`record name,firstname: string[30]; man:boolean end`).

Создание собственного типа данных — запись

При использовании записей очень удобно описать собственный тип данных (в разделе `type`) и потом использовать переменные этого типа.

Пример 15.3. Рекомендуемое описание переменных типа Запись

```
type coord=record
             x,y:real
           end;
     people=record
              name,
              family:string[30];
              man:boolean
            end;
var
   pointA,pointB : coord;
   Vasya : people;
```

Массив записей

Описание собственного типа данных позволяет создать *массив записей*. При обращении к полям элементов такого массива нужно

будет указать имя массива, индекс в квадратных скобках, точку и имя поля, к которому вы обращаетесь.

Пример 15.4. Описание и использование массива записей

```
type coord=record
             x,y:real
           end;
var
   points : array [1..20] of coord;
{ После служебного слова of в описании массива
  нельзя написать record и описать структуру записи.
  Можно только использовать уже описанный тип данных }
   i : integer;
begin
  for i := 1 to 20 do
  begin
    points[i].x := i*5;
    points[i].y := 0
  end
end.
```

Оператор присоединения with

При работе с переменными типа Запись иногда возникает неудобство, связанное с тем, что написание этих переменных оказывается очень длинным. Если вам нужно обратиться к большому количеству полей переменной типа Запись в одном месте программы, вы можете выделить этот блок программы (операторными скобками) и указать перед ним, что к именам полей, употребляемым в блоке, нужно *присоединить* имя переменной типа Запись.

Формат оператора присоединения:

```
with <переменная типа запись> do <оператор, для которого выполнится присоединение>
```

Пример 15.5. Использование оператора присоединения with

```
type coord=record
             x,y:real
           end;
var
   points : array [1..20] of coord;
   i : integer;
```

```
begin
  for i := 1 to 20 do
    with points[i] do
    begin
      x := i*5;
      y := 0
    end
end.
```

Присоединение выполняется не ко всем переменным, находящимся под действием оператора with, а только к тем, которые по имени совпадают с именами полей переменной в операторе with. В примере 15.5 points[i] присоединяется к полям x и y и не присоединяется к переменной i.

Нужно, однако, быть внимательным при использовании имен переменных, совпадающих с именами полей записи. Внутри действия оператора with эти переменные окажутся недоступны.

Пример 15.6. Сокрытие области видимости переменных, имена которых совпадают с именами полей в операторе with

```
type coord=record
             x,y:real
           end;
var
   points : array [1..20] of coord;
   i,x    : integer;
begin
  x := 10;
  for i := 1 to 20 do
    with points[i] do
    begin
      x := x*5; { В обеих частях оператора
                  присваивания обращение происходит
                  к полю points[i].x. К переменной x
                  здесь обратиться нельзя }
      y := 0
    end
end.
```

Как правило, не рекомендуется использовать для переменных и полей одинаковые имена, чтобы избежать возможной путаницы.

Пример выбора структуры данных

Рассмотрим задачу обработки книг, хранящихся в библиотеке.

Так как книг много, имеет смысл хранить их в массиве.

Так как параметры книги разнотипны, имеет смысл создать тип данных «книга», хранящий в себе все параметры книги.

Про книгу необходимо знать: название, автора, издательство, год издания, количество страниц, цену.

Анализируем типы данных. Название, автор и издательство — это строковые данные (`string`). Имеет смысл только ограничить для каждого поля длину, чтобы не хранить слишком много ненужных символов. Год издания и количество страниц — целые числа (`integer`). Цена — количество рублей и копеек. Тут есть два варианта — хранить цену как целое количество копеек и при выводе делить на 100, чтобы получить рубли, или хранить цену как дробное число. Первое немного экономит память. Второе — проще. Выберем для разнообразия `real`.

Вот что у нас получилось:

Пример 15.7. Создание структуры данных для хранения информации о книгах в библиотеке

```
const n=100;
type
 book=record                          { Книга библиотеки       }
        title       : string[40]; { Название               }
        author      : string[25]; { Автор                  }
        publishing  : string[25]; { Издательство           }
        year        : integer;    { Год издания            }
        pages       : integer;    { Количество страниц     }
        price       : real;       { Цена                   }
      end;
var
   books : array [1..n] of book;
```

Чем больше программа, которую вы пишете, тем больше внимания необходимо уделять оформлению кода программы и комментариям. Обратите внимание на оформление описания типа данных в примере 15.7. Каждое поле написано на отдельной строке, рядом с каждым из них написано его назначение. Чем больше программа, которую вы пишете, тем больше времени вы потратите на ее на-

писание. Иногда это занимает несколько дней, недель, месяцев. Чтобы не мучиться через некоторое время, вспоминая, зачем вам была нужна когда-то переменная с именем iii, лучше сразу дать этой переменной осмысленное имя (по которому можно догадаться о ее назначении; например, count_people) и в разделе описания оформить ее смысл так, чтобы он сразу бросался в глаза:

```
var count_people : integer; { Количество
 учеников, зарегистрированных в базе данных }
```

Записи записей

Заметим, можно использовать записи, состоящие из записей, и записи, состоящие из массивов, и массивы записей массивов и пр. В этом случае обращение к конкретному полю может стать весьма длинным.

Пример 15.8. Использование записей и массивов для создания сложных структур данных

```
{ Структура данных компьютерной игры,
  в которой по экрану "ползает" несколько змеек }
const n=100;                 { Наибольшая длина змейки }
type
 point=record         { Координаты одной клетки змейки }
        x,y : integer        { Координаты точки экрана }
      end;
 cells=array[1..n] of point; { Все тело змейки           }
 snake=record                { Вся информация о змейке }
        body   : cells;      { Тело змейки               }
        length : integer;    { Длина змейки              }
        start  : integer;    { Положение головы змейки }
        finish : integer     { Положение хвоста змейки }
      end;
var
 pitons:array [1..5] of snake; { Будет 5 змеек           }
begin
{ Пример обращения к координате "y" начальной клетки
  3-й змейки }
  pitons[3].body[pitons[3].start].y:=10;
end.
```

Выводы

1. Для удобства программирования и последующего анализа программы рекомендуется структурировать данные.
2. Если в программе обрабатывается информация об объекте с несколькими параметрами, рекомендуется создать для этого объекта отдельный тип данных — запись — который будет хранить все параметры объекта.
3. Запись описывается парой операторов `record` и `end`, между которыми перечисляются имена полей записи с указанием их типов данных.
4. Можно создавать массивы записей. При этом нужно обязательно создать свой тип данных — запись.
5. Для сокращения написания переменных типа Запись при обращении к нескольким полям одной переменной используют оператор присоединения `with`.

Контрольные вопросы

1. Зачем нужно использовать записи в программе?
2. Какими операторами оформляются записи?
3. Опишите структуру данных, хранящую информацию о марке автомобиля (название, цена, максимальная скорость, время разгона до 100 км/ч, объем двигателя).
4. Опишите структуру данных, хранящую информацию о нескольких автомобилях из п. 3.
5. Приведите пример обращения к полю «цена» 5-го автомобиля из п. 4.
6. Как можно сократить написание имен переменных типа Запись при обращении к нескольким полям одной переменной?

ТЕМА 16
Динамические переменные

Все способы выделения памяти, которые мы рассматривали ранее (имеется в виду выделение памяти для хранения данных — переменных и постоянных величин) называются *статическими*. Это значит, что объем памяти, которая выделялась для работы нашей программы, был заранее известен к моменту ее (программы) запуска. Эта память единовременно выделялась и существовала все время работы нашей программы. После окончания ее работы память снова освобождалась.

В этой беседе мы рассмотрим другой способ работы с памятью — *динамический*. Память при этом выделяется именно в тот момент, когда становится нужна программе, и освобождается тогда, когда больше не нужна. Этот способ более удобен с точки зрения экономии памяти, но требует больше умственных затрат и аккуратности программиста.

Урок 16.1. Выделение памяти

Прежде чем мы начнем говорить о динамическом выделении памяти, необходимо уточнить, о чем же вообще идет речь — что означает «выделение памяти» и кто этим занимается. Если вам это хорошо известно, перейдите к следующему уроку.

В компьютере имеется два «руководителя», которые управляют работой всего и всех — процессор и операционная система. *Процессор* — аппаратная часть компьютера (hardware). Он выполняет программы и управляет компьютером на уровне электрических сигналов. *Операционная система* — часть программного обеспечения компьютера (software). Причем самая главная ее часть. Самая главная программа, которая запускается практически сразу после включения компьютера и управляет всеми ресурсами компьютера, а также общается с пользователем (человеком). Без

операционной системы компьютер работать не может (точнее, работать-то он будет, но пользоваться им будет практически невозможно).

Любая программа, которую выполняет процессор, должна находиться в *основной памяти* компьютера. Ее назначение — хранение программ, выполняющихся в конкретный момент времени процессором, и данных, которые эти программы обрабатывают. Это микросхемы, которые в современных компьютерах расположены внутри, на материнской плате. Основная память разделяется на постоянную (ПЗУ, ROM) и оперативную (ОЗУ, RAM). Нас будет интересовать ОЗУ — оперативная память компьютера (ПЗУ нужна для первоначального включения компьютера, в нее нельзя ничего записать).

Когда вы запускаете любую программу (неважно, написанную вами или чужую), она сначала загружается в оперативную память (обычно с диска), потом ей выделяется память (статическая), которую программа запрашивает в момент запуска, потом начинает выполняться сама программа. Когда программа заканчивает свою работу, память, выделенная программе для данных, и память, выделенная для хранения в ОЗУ самой программы, освобождаются.

Всеми этими процессами — загрузкой программы в ОЗУ, выделением памяти для тела программы и для ее данных, завершением программы и освобождением памяти (и еще много чем другим) — управляет операционная система (ОС). Программа как бы «просит» операционную систему выделить ей память, и ОС выделяет ее программе (если такое количество свободной памяти в данный момент имеется).

«Выделяет» — означает, что у ОС имеется специальная таблица (список), где перечислены все свободные блоки памяти (их адреса и размеры). ОС ищет в этой таблице блок подходящего размера, сообщает его адрес программе и корректирует таблицу в соответствии с занятым фрагментом. Когда программа заканчивает свою работу, освобожденный блок памяти добавляется в список свободных блоков.

Если операционная система однозадачная (как MS-DOS, например), программа может рассчитывать на всю оперативную память, которая имеется в данный момент в компьютере (или почти всю). Если многозадачная — память распределяется между несколькими программами.

Среда программирования Turbo Pascal написана под операционную систему MS-DOS. Ввиду некоторых особенностей работы этой ОС, если не использовать дополнительных менеджеров памяти, в своих программах на Паскале вы можете рассчитывать только на 64 Кбайт (65 536 байт) оперативной памяти (именно столько может адресовать 16-разрядный процессор). И хотя у вас почти наверняка памяти гораздо больше, и процессор гораздо лучше, вы не сможете (без специальных программ) создать, например, целочисленный массив из 40 000 элементов (40 000 по 2 байта на каждый элемент — это больше, чем 65 536).

Если вам нужно работать с данными большего размера, имеет смысл подумать о другой, более современной среде программирования (например, Borland Delphi). Язык программирования тот же (Паскаль), а возможностей больше.

Урок 16.2. Адреса

Для хранения информации о том, где в памяти находится место, куда вы собираетесь класть свои данные, используются *адреса*. Это целые числа (в зависимости от ОС и процессора, 16-, 32- или 64-разрядные). Они соответствуют номеру ячейки памяти (байта), начиная с которого хранятся ваши данные.

Если вы написали в вашей программе: `var a,b:integer`, это означает, что вы просите выделить вам 2 ячейки памяти, достаточные для хранения двух чисел типа `Integer` (в Turbo Pascal это 2 байта, в Delphi — 4). При этом компилятор (программа, которая переводит вашу программу с языка Паскаль на язык процессора) записывает себе в специальную таблицу относительные адреса для ячейки памяти `a` и для ячейки `b`.

ЗАМЕЧАНИЕ

Относительные — потому что компилятор не знает, какие конкретно адреса памяти выделит для вашей программы операционная система. Но он знает, сколько всего памяти требуется программе. И он запоминает в отдельной таблице, что ячейка `a` *будет находиться, например, по адресу «+4» относительно начала блока памяти, выделенного программой, а ячейка* `b` *— по адресу «+6». Когда программа запустится, к адресу начала выделенного блока прибавится запомненное смещение, и программа будет обращаться уже к конкретному адресу ОЗУ.*

Эти адреса записаны прямо в теле программы. Места хранения ячеек a и b не меняются за все время ее выполнения.

Но есть возможность хранить эти адреса в отдельных ячейках памяти. Смысл в этом, правда, появляется только, если вы собираетесь управлять памятью более хитро, нежели просто использовать обычные переменные.

Итак, заранее предупреждаем — то, что мы сейчас расскажем, само по себе не имеет никакого практического смысла. Это хорошо только в случае «хитрого» использования. Но чтобы научиться делать что-то «хитрое», нужно сначала овладеть технологией.

Урок 16.3. Указатели

Указатели на отдельные переменные

Для хранения адресов ячеек памяти используются переменные-*указатели*. Как и обычные переменные, они описываются в разделе var. Формат описания:

```
<имя переменой>: ^<имя типа данных, на переменные которого будет указывать указатель>;
```

Для указания того, что переменная является указателем, перед именем типа данных нужно поставить символ ^ («шляпка», Shift+6 на клавиатуре). Автор Паскаля предполагал использовать для этой цели символ «стрелочка» (↑), чтобы подчеркнуть, что он на что-то указывает. Но такого символа на клавиатуре нет, поэтому используют как бы стрелочку без вертикальной палочки. Пример:

```
var p:^integer;
```

Это означает, что переменная p способна хранить в себе адрес ячейки памяти целого типа данных.

ПРИМЕЧАНИЕ

С точки зрения процессора, нет никакой разницы между адресом целочисленной ячейки памяти и адресом любой другой, например вещественной. Во всех случаях хранение адреса требует одинаковое количество байт. Но программы с указателями потенциально содержат в себе гораздо больше ошибок, чем обычные. Автор Паскаля намеренно требует указывать, на ячейку какого типа данных ссылается указатель, чтобы хотя бы часть ошибок можно было «отловить» на этапе компиляции (эти ошибки возникают

из-за того, что указатели по вине программиста указывают вовсе не на то, на что программист предполагает).

Заметим, что в приведенном примере никакого динамического выделения памяти, которого вы, вероятно, давно ожидаете, вовсе нет. Память выделяется статически, в момент запуска программы. Мы пока только показали, как можно хранить в программе адрес какой-то ячейки памяти.

Основная идея этого заключается в динамическом выделении памяти. Вы ведь помните, что означает «выделение памяти»? Если память будет выдаваться нашей программе не сразу, а в какой-то другой момент времени, операционная система сообщит ей адрес выделенной ячейки. А его ведь нужно будет где-то сохранить, чтобы впоследствии к этой памяти обращаться! При статическом выделении памяти этим занимался компилятор. Он сохранял адреса всех переменных и подставлял эти адреса во все те места программы, в которых мы к переменным обращались. А при динамическом выделении памяти компилятор не может нам помочь — он ведь не знает адреса, который сообщит нам операционная система. Значит, нужно самим предусмотреть место, где этот адрес можно будет хранить. Преимущественно для этого и нужны переменные-указатели.

Для того чтобы в переменную-указатель положить адрес статической ячейки памяти (а мы пока с другими работать и не умеем), используется символ `@` («собачка», коммерческое *AT*, Shift+2). Пример: `p:=@count;` (если `p` описан как указатель на тот же тип данных, что и `count`, то переменная `p` будет содержать адрес ячейки памяти, в которой хранится переменная `count`).

Чтобы обратиться к содержимому ячейки памяти, на которую ссылается переменная-указатель, используется тот же символ ^ («шляпка»), стоящий после имени переменной-указателя. Пример: `p^:=10;` (положить в ячейку, на которую указывает `p`, число 10).

Пример 16.1. Простейшее использование переменных-указателей

```
var
 vasya,petya : integer; { Две целочисленные переменные }
 p,q         :^integer; { Два указателя на integer     }
begin
 vasya := 10;
```

```
 p := @vasya; { Указатель p содержит адрес
                переменной vasya }
 q:=p;        { Указатели можно присваивать друг другу }
 q^:=15;      { Переменная vasya хранит число 15,
                так как q указывает на ту же ячейку,
                что и p, а p указывает на vasya }
 p^:=q^+5;    { Переменная vasya увеличилась на 5 }
end.
```

Указатели на блоки переменных

К переменным-указателям применимы операции inc и dec. Они увеличивают и уменьшают значение указателя. Но не на 1 (как можно было бы предположить), а на то количество, которое занимает в памяти переменная того типа данных, на который ссылается указатель. Это свойство бесполезно при работе с отдельными переменными, но его можно использовать для обращения к элементам массива.

Пример 16.2. Использование операции приращения указателя для обращения к элементам массива

```
var
 a : array [1..20] of integer;
 p :^integer;
 i : integer;
begin
  p:=@a[1]; { Положим в переменную-указатель p
              адрес первой ячейки массива }
{ Переберем все ячейки массива }
  for i:=1 to 20 do
  begin
    p^:=i*5; { В текущий элемент массива положим число }
    inc(p)   { Переменная p ссылается на следующий
               элемент массива }
  end
end.
```

Урок 16.4. Динамическое выделение памяти

В Паскале существует два способа динамического выделения памяти — простой и «продвинутый». В исходном Паскале был

предусмотрен только простой способ. В современных Паскалях доступны оба.

New и Dispose

Первый способ осуществляется парой процедур `New` и `Dispose`. Первая выделяет память, вторая — освобождает. В обеих процедурах требуется один параметр — переменная-указатель.

Процедура `New` выделяет область памяти такого размера, чтобы в нее поместился объект того типа данных, на который может указывать указатель. После выполнения процедуры `New` указатель содержит адрес созданного объекта. Пример: `New(p)`. Как вы понимаете, при этом происходит передача параметра по адресу. То есть параметром процедуры `New` может быть только переменная и не может быть выражение.

Процедура `Dispose` освобождает область памяти, выделенную предварительно процедурой `New` для объекта, на который ссылается указатель. Пример: `Dispose(p)`. Указатель `p` должен к этому моменту ссылаться на предварительно выделенную область памяти. Если это не так — последствия процедуры `Dispose` не определены. Скорее всего, ваша программа «вылетит» с сообщением об ошибке.

После выполнения процедуры `Dispose` значение переменной-указателя считается неопределенным. Попытка прочитать данные, на которые указывает указатель, или записать туда новые данные может иметь непредсказуемый результат. Хотя, скорее всего, значение переменной-указателя при этом не изменится. И сами данные не изменятся. Но эта область памяти уже не будет считаться доступной вам. Это можно себе представить, как если бы вы жили в какой-то квартире, а потом продали ее и уехали, но оставили себе при этом ключи. С одной стороны, вы имеете возможность (может быть) в квартиру войти и даже что-то из нее забрать. С другой стороны, последствия вашего поступка при этом будут непредсказуемы.

Пример 16.3. Использование процедур New и Dispose

```
var a,vasya : integer;
    q,p     :^integer;
begin
  new(p);     { Создаем новую переменную типа integer.
                Переменная p содержит ее адрес }
```

```
  p^:=10;      { Новая переменная хранит число 10 }
  q:=p;        { Запомнили ее адрес еще раз }
  new(p);      { Создали еще одну переменную.
                 Нужно понимать, что после процедуры New
                 переменная-указатель изменит свое
                 значение. Если бы перед этой строкой
                 мы не запомнили адрес переменной,
                 созданной ранее, то он оказался
                 бы утерян. Переменная продолжала бы
                 существовать, но к ней нельзя было бы
                 обратиться и выделенную для нее память
                 нельзя было бы освободить.
                 Необходимо следить, чтобы адреса
                 динамических переменных не терялись! }
  p^:=3;       { Вторая переменная хранит число 3 }
  dispose(p);  { Память, выделенная под вторую
                 переменную, освобождается.
                 Значение “3” пропадает }
  p:=q;        { Переменная p снова содержит адрес
                 первой переменной }
  dispose(p);  { Освобождаем память, выделенную для
                 первой переменной }
  writeln(q^);{ Неосмотря на то, что переменная q
                 все еще содержит адрес ячейки,
                 где хранилось число 10, обращаться
                 к этой ячейке запрещено. Она чужая
                 для нашей программы.
                 Трогать чужое нельзя! }
end.
```

Вероятно, вы обратили внимание, что адреса всех ячеек памяти, которые мы выделяли процедурами New, необходимо где-то хранить. Например, выделять для каждой переменную-указатель. Но при этом получается, что для хранения в памяти переменной требуется меньше места, чем для выделения памяти для хранения указателя для нее (например, в Turbo Pascal для Integer нужно 2 байта, а для указателя — 4). В этом случае использование динамических переменных не экономит память, а еще больше тратит ее.

Если не использовать специальные структуры данных, которые мы будем обсуждать в следующем уроке, то динамическое выделение памяти нужно только в случае одновременного выделения больших объемов памяти (преимущественно массивов).

Динамическое выделение памяти для массивов

Рассмотрим пример. В программе нам требуется использовать два массива очень большого размера. Такого, что каждый такой массив едва помещается в имеющуюся оперативную память, а сразу оба — не поместятся. Но каждый из этих массивов нужен нам не одновременно, а по очереди — сначала один, потом другой. Вы скажете, ну и используем один и тот же массив для разных целей! Это можно сделать, только если в них хранятся однотипные данные. Как же быть?

Сделаем так. Опишем два типа данных для каждого вида массивов, и опишем переменные-указатели на эти массивы. В начале программы создадим один из массивов, поработаем с ним, освободим выделенную для него память, потом выделим память под второй массив.

Пример 16.4. Попеременное использование большого объема памяти для хранения разных данных

```
type masA=array[1..30000] of integer;
     masB=array[1..60000] of char;
var a : ^masA; { Указатель должен ссылаться на тип
                 данных, имеющий имя. Нельзя написать
                 ^array[1..30000] of integer; }
    b : ^masB;
begin
{ ...Здесь делаем какие-то начальные действия... }
  new(a);     { Выделяем память под первый массив }
{ Здесь работаем с элементами этого массива.
  Для обращения, например, к пятому элементу
  нужно написать a^[5] }
  dispose(a); { Освобождаем память,
              занятую первым массивом }

  new(b);     { Выделяем память под второй массив }
{ Здесь работаем с элементами второго массива }
  dispose(b); { Освобождаем память,
              занятую вторым массивом }
end.
```

ЗАМЕЧАНИЕ

Термином «динамический массив» в настоящее время принято называть не то, что мы только что продемонстрировали (массив, под хранение

которого память выделена динамически), а массив неопределенной длины. Применение подобных массивов нами не одобряется, и их изучение выходит за рамки нашего курса. В Turbo Pascal они не допускаются и возможны только в Delphi. Вкратце, такие массивы описываются, например, так: `a:array of integer`. *Для указания длины, которую нужно выделить для ячеек массива, используют процедуру* `setlength`. *У нее первый параметр — имя массива, второй — длина. Например,* `setlength(a,20)`.

ЗАМЕЧАНИЕ

Возможно, вас давно уже мучает резонный вопрос: зачем вообще нужно выделять память под массивы динамически? Не проще ли заранее зарезервировать столько памяти, сколько можно, и с ней работать?! Ведь больше памяти все равно использовать нельзя!

Есть некоторая логика в этом утверждении. Но для современных компьютеров динамическое выделение памяти становится все более актуальным. Ведь почти всегда вы работаете в многозадачной операционной системе. И это значит, что одновременно с программой, которую написали вы, работает еще несколько программ. Например, играет музыка, работает антивирус и из Интернета скачиваются фотографии ваших друзей. А ваша программа — обычная записная книжка, в которой вы храните нужные телефоны. Спрашивается: какой объем памяти должна зарезервировать программа для своей работы!? Если заранее описать массив из небольшого числа элементов, их может в какой-то момент не хватить. А если заказать большое число элементов, во-первых, их все равно когда-нибудь может не хватить, во-вторых, окажется, что большой объем памяти обычно не используется. Согласитесь, было бы не очень хорошо, если записная книжка при старте резервировала под себя почти всю память, и это мешало бы работе других программ! Значит, правильнее всего резервировать столько памяти, сколько нужно в настоящий момент. Понадобится увеличить — попросим больше.

GetMem и FreeMem

Гораздо более гибкий (и более трудоемкий с точки зрения работы программиста) способ управления динамической памятью предлагают нам процедуры `GetMem` и `FreeMem`. Как и в случае с парой `New-Dispose`, первая процедура выделяет память динамически, вторая — освобождает. Но в отличие от `New` и `Dispose`, которые сами

понимали, какого размера нужно выделить память и какого размера блок памяти нужно освободить, для процедур GetMem и FreeMem это должен указать программист.

Формат использования:

```
GetMem(<переменная-указатель>,<объем памяти в байтах>);
FreeMem(<переменная-указатель>,<объем памяти в байтах>);
```

При этом переменная-указатель может быть описана как указатель на любой тип данных. Программист также должен заботиться о том, чтобы заказываемого объема данных хватило для переменных этого типа. Чтобы не ошибиться в количестве байт, обычно используют функцию sizeof. Ее аргументом нужно указать имя типа данных; ее результатом является число байт, требуемых для хранения переменной этого типа данных.

Обычно парой GetMem-FreeMem пользуются для динамического выделения памяти под массивы, длина которых заранее не известна. Так, например, для динамического выделения памяти под целочисленный массив из 300 элементов разумно написать так: GetMem(a,SizeOf(integer)*300), где переменная a — указатель на integer (a:^integer).

Обращение к элементам массива, созданного динамически

Неудобством этого способа является не очень прямой доступ к элементам такого массива. Ведь описав обычный массив, можно обратиться к его произвольному элементу путем индексации: например для обращения к десятому элементу массива mas достаточно написать mas[10]. А к десятому элементу только что рассмотренного массива a так просто не обратишься. Перебрать по порядку все элементы легко. Это можно сделать точно таким же способом, как в примере 16.5.

Пример 16.5. Последовательное обращение к элементам динамически созданного массива

```
var
 a,p : ^integer;
 i   : integer;
begin
  GetMem(a,sizeof(integer)*300); { Выделили память
                                   под массив }
```

```
{ Переберем все ячейки массива }
  p := a; { Чтобы не потерять адрес начала массива,
            для обращения к его элементам используем
            дополнительную переменную-указатель }
  for i:=1 to 300 do
  begin
    p^:=i*5; { В текущий элемент массива положим число }
    inc(p)   { Переменная p ссылается на следующий
               элемент массива }
  end;
  FreeMem(a,sizeof(integer)*300) { Освободили память,
                                   выделенную ранее
                                   под массив }
end.
```

Если нужно получить прямой и быстрый способ обращения к указанной ячейке такого массива, можно использовать процедуру inc с двумя аргументами. Первый аргумент — переменная-указатель, хранящая адрес начала такого массива. Второй аргумент — на сколько нужно «сдвинуться» относительно первого элемента. Например, для обращения к десятому элементу можно написать так:

```
q:=a; inc(q,9);
```

Теперь q^ — интересующий нас элемент.

Массив переменной длины

Эффект массива переменной длины можно создать самостоятельно, используя процедуры GetMem-FreeMem. Рассмотрим самый типичный случай — необходимо сохранить в памяти данные, вводимые с клавиатуры, количество которых заранее неизвестно. При появлении каждого нового элемента будем увеличивать размер выделенной памяти на 1. Конечно же, имеющиеся на данный момент элементы массива придется переписывать во вновь выделенную память.

Пример 16.6. Массив переменной длины

```
var
 a :^integer;     { Указатель на хранимый массив }
 b :^integer;     { Временная переменная для
                    указателя на новый массив }
```

```
  pa,pb :^integer; { Временные переменные-указатели для
                     перемещения по элементам массивов }
  x : integer;     { Вводимое число }
  n : integer;     { Размер массива }
  i : integer;     { Счетчик цикла }
begin
  n:=0;            { Начальный размер массива }
  readln(x);       { Читаем первое число }
  while x<>0 do    { Будем сохранять числа, пока
                     вводимое число не равно нулю }
  begin
    inc(n);        { Новая длина массива }
    { Текущий массив имеет размер на 1 меньше,
      чем требуется. Выделим нужное число ячеек: }
    GetMem(b,sizeof(integer)*n);
    pb:=b;         { Установим указатель pb
                     на первый элемент массива b }
    { Перепишем все элементы массива a (n-1 штук)
      в массив b: }
    if n>1 then    { Если нужно переписывать элементы }
    begin
      pa:=a;       { Установим указатель pa
                     на первый элемент массива a }
      for i:=1 to n-1 do { Переберем все элементы }
      begin
        pb^:=pa^;  { Копируем элементы из a в b }
        inc(pa);   { Перемещаемся на следующий
                     элемент массива a }
        inc(pb)    { Перемещаемся на следующий
                     элемент массива b }
      end
    end;
    pb^:=x;        { Запишем введеное число x
                     последним элементом массива b }
    { Освободим память, занятую старым массивом a: }
    FreeMem(a,sizeof(integer)*(n-1));
    { Установим указатель массива a на массив b: }
    a:=b;
    readln(x)      { Прочитаем следующее число }
  end
end.
```

Вы, конечно, заметили недостаток, приведенного примера. При добавлении каждого нового элемента приходится переписывать за-

ново всю предыдущую часть массива. Можно избавиться от этого недостатка, если выделять память не каждый раз, а размерами, равными степеням двойки. То есть если в какой-то момент размер массива равен, например, 8 и требуется добавить еще один элемент, нужно выделить сразу 16 ячеек. В этом случае в последующие 7 добавлений (если они потребуются) не нужно будет переписывать предыдущую часть массива. Памяти для такого метода потребуется не более чем в 2 раза больше, чем нужно для данных.

Пример 16.7. Массив переменной длины с линейной сложностью добавления элементов

```
var
 a :^integer;     { Указатель на хранимый массив }
 b :^integer;     { Временная переменная для
                    указателя на новый массив }
 pa,pb :^integer; { Временные переменные-указатели для
                    перемещения по элементам массивов }
 x : integer;     { Вводимое число }
 n : integer;     { Размер памяти,
                    выделенной для массива }
 k : integer;     { Количество занятых ячеек массива }
 i : integer;     { Счетчик цикла }
begin
  n:=1;           { Начальный размер массива }
  k:=0;           { Число занятых ячеек массива }
  GetMem(a,sizeof(integer)); { Память под одну ячейку }
  readln(x);      { Читаем первое число }
  while x<>0 do   { Будем сохранять числа, пока
                    вводимое число не равно нулю }
  begin
    inc(k);       { Новая длина массива }
    { Проверим, помещается ли новое число в выделенную
      память или требуется выделить новую }
    if k>n then   { Не помещается }
    begin
      n:=n*2;     { Удваиваем требуемую память }
      { Выделим память под новый массив }
      GetMem(b,sizeof(integer)*n);
      { Перепишем все элементы массива a (k-1 штук)
        в массив b }
      pa:=a;      { Установим указатель pa
                    на первый элемент массива a }
```

```
      pb:=b;      { Установим указатель pb
                    на первый элемент массива b }
      for i:=1 to k-1 do { Переберем все элементы }
      begin
        pb^:=pa^; { Копируем элементы из a в b }
        inc(pa);  { Перемещаемся на следующий
                    элемент массива a }
        inc(pb)   { Перемещаемся на следующий
                    элемент массива b }
      end;
      pb^:=x;     { Запишем введеное число x
                    последним элементом массива b }
      { Освободим память, занятую старым массивом a }
      FreeMem(a,sizeof(integer)*(k-1));
      { Установим указатель массива a на массив b }
      a:=b
    end
    else  { Если в массиве a есть еще место }
    begin
      { Переместим указатель pa на первый
        незанятый элемент массива a }
      pa:=a;
      inc(pa,k-1);
      pa^:=x       { Запишем введеное число x }
    end;
    readln(x)     { Прочитаем следующее число }
  end
end.
```

Выводы

1. При объявлении переменных в разделе `var` память для них выделяется в момент запуска программы и занята все время работы программы. Это называется *статическим выделением памяти*.
2. Для хранения информации об адресе переменной используют переменные-указатели.
3. Для указания того, что переменная является указателем, перед именем типа данных, на который создается указатель, ставят знак ^ («шляпка»).

4. Для создания указателя на нестандартную ячейку памяти (массив или запись) необходимо создать тип данных, имя которого указывается после ^ при описании переменной-указателя.
5. Для обращения к той ячейке, адрес которой хранится в переменной-указателе, ставят знак ^ после имени переменной-указателя.
6. Для создания динамической переменной (выделения под нее памяти) используют процедуры `New` или `GetMem`.
7. Для уничтожения динамической переменной (освобождения занимаемой ею памяти) используют процедуры `Dispose` или `FreeMem`.

Контрольные вопросы

1. В чем отличие динамического выделения памяти от статического?
2. Кто управляет использованием памяти компьютера?
3. Как в программе создать переменную-указатель?
4. Как обратиться к ячейке памяти, на которую ссылается переменная-указатель?
5. Как переместить переменную-указатель на следующую ячейку памяти?
6. Как в программе выделить память под динамическую переменную?
7. Как освободить память, выделенную под динамическую переменную?
8. Для программы:

```
var a:integer; p:^integer;
begin        { Момент 0 }
  a:=10;     { Момент 1 }
  new(p);    { Момент 2 }
  p^:=15;    { Момент 3 }
  dispose(p);{ Момент 4 }
  writeln(a);{ Момент 5 }
end.         { Момент 6 }
```

а) в какой момент выделяется память для переменной `a`?

б) в какой момент освобождается память, выделенная для переменной a?

в) в какой момент выделяется память для переменной p^?

г) в какой момент освобождается память, выделенная для переменной p^?

ТЕМА 17

Динамические структуры данных. Стек

Самый удобный способ выделения памяти — когда ее выделяется ровно столько, сколько в каждый конкретный момент времени требуется программе. В этой теме мы начинаем рассказ о том, какими способами можно приблизиться к этому волшебному состоянию. «Приблизиться» в данном случае значит, что программа будет требовать памяти почти ровно столько, сколько ей нужно, плюс небольшие «накладные расходы».

Как вы, вероятно, уже поняли, основная проблема состоит не в том, чтобы выделить памяти ровно столько, сколько нужно. Это как раз просто: понадобился нужный «кусок» — попросил — дали. Проблема в том, что при каждом выделении памяти нужно хранить адрес этого самого выданного «куска». «Кусков» много. А так как количество «кусков» заранее неизвестно, создать заранее массив для адресов не получается (при объявлении массива должен быть известен его размер). Предлагаемый метод хранит информацию в виде цепочки. Каждый раз, когда выделяется новый «кусок» памяти, в нем предусмотрено место для хранения адреса следующего «куска». Для реализации этой идеи достаточно описать только одну статическую переменную — адрес начала такой цепочки, состоящей из объектов вида: данные + адрес следующего объекта.

Урок 17.1. Опишем тип данных

Можно представить себе эту структуру данных в таком виде. В области данных хранится то, что требуется. В области адреса хранится ссылка (адрес) следующего блока. Последний элемент этой конструкции должен хранить «терминатор» — некий признак того, что он последний и что за ним больше элементов нет. Обычно для этого используют поле адреса — помещают в него такой адрес, ко-

торого быть не может. В языке Паскаль для примерно таких нужд имеется специальное служебное слово `nil`. Оно означает «адрес, которого нет», «ссылка на несуществующую область памяти». Это значение и кладут в ячейку-адрес (рис. 17.1).

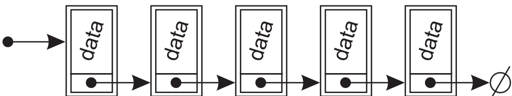


Рис. 17.1. Организация простейшей динамической цепочки данных

```
type MyElement=record
        data:<Тип данных, которые храним>;
        next:<Указатель на точно такой же
              элемент данных, как этот>
     end;
```

Для простоты предположим, что каждый хранимый элемент данных — строка. Тогда получается, что конструкция должна быть описана следующим образом:

```
type MyElement=record
        data:string;
        next:^MyElement
     end;
```

Но язык Паскаль не позволяет описать такую конструкцию. Компилятору такое не нравится — он воспринимает это как попытку сослаться (создать поле-ссылку) на тип данных, который еще не описан (то есть описание которого еще не закончено). Нужно пойти более хитрым путем: описать тип данных — указатель на конструкцию, описание которой будет приведено ниже. А в этой конструкции использовать этот тип данных — указатель.

Пример 17.1. Описание простейшей динамической структуры данных

```
type
 pElement=^Element; { Создаем указатель на тип данных,
                      который будет описан позже.
                      Так как все указатели в памяти
                      все равно занимают одинаковое
                      количество байт, такое
                      "впередссылающееся" описание
                      компилятор не смущает }
 Element=record
```

```
    data : string;      { Здесь можно использовать любой
                          тип данных. Можно описать
                          несколько полей }
    next : pElement     { Это ссылка на следующий элемент
                          структуры данных }
  var
    pBegin : pElement; { Достаточно выделить только одну
                          статическую переменную, чтобы
                          можно было хранить цепочку
                          неопределенной длины }
```

Обратите внимание: элемент нашей структуры данных — запись. Мы описываем тип данных только одного элемента. Мы не описываем тип данных для всей конструкции. Конструкция состоит из однотипных элементов, связанных между собой. С точки зрения Паскаля это просто описание одной переменной. При создании каждого нового элемента Паскалю нет никакого дела, что за структура данных у нас будет получаться. То, что элементы будут соединяться друг с другом, «нанизываясь на цепочку», — дело рук программиста.

Урок 17.2. Создание стека и основные операции со стеком

Мы нарисовали общую конструкцию. В таком виде эта структура данных (когда элементы связаны друг с другом в одну цепочку и каждый элемент хранит адрес следующего) называется *однонаправленным списком*.

Рассмотрим частный случай однонаправленного списка — когда операции добавления и удаления элементов разрешены только с одной стороны. На языке программистов такая конструкция называется LIFO («last in — first out», что означает «последним пришел — первым уйдешь»). По-другому это можно себе представить как стопку книг. Добавлять в нее книги можно только сверху, и забирать из нее книги можно только сверху. Последняя книга, положенная в стопку, будет забрана из нее первой. Такая структура данных называется *стек* (стопка).

Описание типов данных, которые нужны для работы со стеком, такое же, как только что описанное нами. Проще всего это обозначить таким образом:

```
type pStack=pElement;
```

На тот случай, если вы не поняли, каким при этом получилось описание структуры данных, приведем его полностью:

```
type pStack=^Stack;
     Stack=record
        data:string;
        next:pStack
     end;
```

В разделе описания переменных для описания стека нужна одна-единственная переменная — адрес последнего элемента, который положен в стек. Он называется *вершина стека*. Его описание будет выглядеть так:

```
var pTop:pStack=nil;
```

Мы добавили к имени `Top` (вершина) букву `p`, чтобы лишний раз показать, что эта переменная является указателем. Также в приведенном примере мы сразу задали начальное значение переменной `pTop (nil)`. Это значит, что в момент старта программы стек пуст (в нем пока ничего не хранится).

Вообще говоря, такое задание начальных значений переменных возможно не во всех компиляторах Паскаля. Если в используемой вами версии это невозможно (компилятор выдает ошибку), задайте начальное значение переменной `pTop` первой инструкцией программы:

```
var pTop:pStack;
begin
  pTop:=nil;
```

Как мы уже говорили, при работе со стеком допустимы только две операции: положить элемент на стек и взять верхний элемент со стека. Опишем эти действия в виде подпрограмм. Имена для этих подпрограмм дадим в соответствии со сложившейся традицией: помещение элемента на стек называется `Push`, изъятие элемента с вершины стека называется `Pop`.

Добавление элемента в стек (Push)

Начнем с реализации подпрограммы `Push`. Для начала уясним, процедура это или функция. Результатом работы этой подпрограммы должно быть действие помещения указанного значения

в стек. Значит, никакого значения подпрограмма Push возвращать не должна. То есть Push — процедура. А так как для своей работы она должна «знать», что нужно положить на стек, то это процедура с параметром. Так как параметр в результате работы процедуры изменяться не должен, передача параметра происходит по значению.

Получаем описание заголовка:

```
procedure Push(chto:string);
```

Здесь chto — то значение, которое мы кладем на стек. Так как при описании типа данных Stack мы договорились, для определенности, что в стеке мы будем хранить строки, то и тип данных параметра chto — строка.

Представим себе состояние стека к моменту, когда мы собираемся положить на его вершину новый элемент (рис. 17.2).

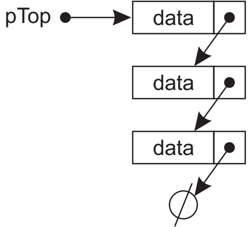


Рис. 17.2. Пример состояния стека перед тем, как в него будет положен еще один элемент

Для начала нужно создать в памяти место, в котором будет храниться новый элемент. Это должна быть процедура new, в качестве параметра которой нужно сообщить переменную-указатель на такой тип данных, под который собираемся выделить память. Выделять память собираемся под элемент стека (Stack). На него умеет ссылаться переменная типа pStack. Нам нужно, чтобы адрес вновь создаваемой ячейки где-то сохранился. Для этого в процедуре Push создадим специальную вспомогательную переменную типа pStack:

```
var p:pStack;
```

Теперь можно создать новую ячейку памяти:

```
new(p);
```

У этой ячейки два поля: поле данных и поле-указатель на следующий элемент.

Заполнить поле данных просто — в нем должны храниться те данные, которые передаются в качестве значения, помещаемого на стек. То есть значение параметра chto:

```
p^.data:=chto;
```

Обратите внимание на то, как мы обращаемся к полю данных: переменная p указывает на нужную нам ячейку памяти. Чтобы обратиться к значению этой ячейки, мы используем знак ^, то есть p^. А теперь нужно обратиться к полю data этой записи. Значит, нужно поставить точку и имя поля. Все вместе получается: p^.data.

Теперь нужно заполнить значение второго поля — адрес следующей ячейки стека. Так как наша новая ячейка должна оказаться верхней, сама она должна ссылаться на следующую за ней ячейку. Для того нужно где-то взять ее адрес. Если мы посмотрим на структуру стека (см. рис. 17.2), то поймем, что на этот элемент в данный момент как раз указывает переменная pTop. Значит, в поле next нужно положить ее значение:

```
p^.next:=pTop;
```

Вроде бы оба поля вновь созданного элемента стека мы заполнили. Нужно ли сделать что-нибудь еще? Посмотрим внимательно на текущую структуру стека (рис. 17.3).

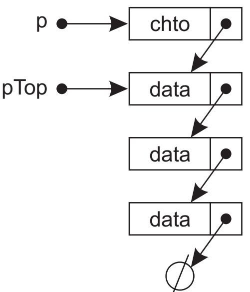


Рис. 17.3. Состояние стека после добавления нового элемента (указатель на вершину пока еще ссылается на прежнюю вершину)

Надеюсь, вы обратили внимание, что в ней указатель на вершину стека (pTop) все еще указывает на прежнюю вершину стека.

Значит, остается переставить его на вновь созданный элемент. Его адрес храниться в переменной p. Получаем:

```
pTop:=p
```

Сведем все сказанное вместе:

Пример 17.2. Процедура Push (добавление элемента на вершину стека)

```
procedure Push(chto:string);
var p : pStack;  { Вспомогательная переменная }
begin
  new(p);         { Выделяем память под новый элемент }
  p^.data:=chto; { Записываем в него сохраняемое значение }
  p^.next:=pTop; { Задаем указатель на следующий }
  pTop:=p         { Делаем новый элемент вершиной стека }
end;
```

Заметим, что мы написали процедуру Push, рассуждая о стеке, в котором уже есть хотя бы один элемент. А еще нужно рассмотреть случай, когда стек пуст. Однако этот случай отличается от непустого стека только тем, что переменная pTop вместо адреса верхнего элемента стека хранит значение nil. Рассмотрев только что написанную процедуру Push, можно убедиться, что в случае пустого стека она создаст новый (и единственный) элемент стека и запишет в его поле-указатель на следующий элемент текущее значение pTop, то есть nil. Но в случае пустого стека именно это и нужно сделать — записать nil в качестве ссылки на второй элемент стека. Значит, наша процедура Push одинаково работает в обоих случаях.

Извлечение элемента из стека (Pop)

Вторая подпрограмма работы со стеком (Pop) должна извлекать верхний элемент стека. При этом этот элемент должен из стека удаляться, а его значение должно возвращаться. Значит Pop — функция. Так как никакой дополнительной информации при вызове мы ей не сообщаем, параметров у нее нет. А тип данных возвращаемого значения должен совпадать с типом данных того, что хранится в стеке. В нашем случае — строка:

```
function Pop:string;
```

Самое простое, что должна сделать функция Pop, — вернуть значение поля данных верхнего элемента стека. На него указывает переменная pTop, поэтому действие очевидно:

```
Pop:=pTop^.data;
```

Осталось только удалить этот элемент из стека. Но под действием «удалить» мы должны понимать два действия: во-первых, нужно очистить память, которую этот элемент занимает, во-вторых, нужно изменить стек так, чтобы этого элемента в нем больше не было. Оба действия сами по себе просты: для очистки памяти нужно вызвать dispose(pTop), а для того, чтобы этого элемента в стеке больше не было, нужно переместить указатель стека pTop на следующий элемент (его адрес хранится в поле next верхнего элемента) — pTop:=pTop^.next.

Но оказывается, что вместе эти два действия вступают в противоречие. Если сначала освободить память, занимаемую верхним элементом, потом нельзя будет взять из него адрес следующего элемента стека. А если сначала взять из памяти верхнего элемента адрес следующего элемента и переместить указатель pTop, потеряется адрес ячейки памяти, которую нужно освободить. Чтобы разрешить это противоречие, нужно, например, запомнить адрес верхнего элемента в отдельной ячейке.

Пример 17.3. Функция Pop (удаление элемента с вершины стека)

```
function Pop:string;
var p : pStack;  { Вспомогательная переменная }
begin
  p:=pTop;      { Запоминаем адрес вершины стека }
  Pop:=p^.data; { Извлекаем данные из верхнего элемента}
  pTop:=p^.next;{ Перемещаем указатель вершины стека
                  на следующий элемент }
  dispose(p)    { Освобождаем память }
end;
```

Проверка стека на пустоту (StackIsEmpty)

Еще одна подпрограмма нужна для работы со стеком — функция, которая проверяет стек на пустоту (что в стеке нет ни одного элемента). Это очень просто проверить — убедиться, что указатель на вершину стека равен nil:

Пример 17.3. Функция StackIsEmpty (проверка стека на пустоту)

```
function StackIsEmpty:boolean;
begin
  StackIsEmpty := pTop=nil
end;
```

Урок 17.3. Использование стека

В качестве примера применения стека приведем программу перевода натурального числа в другую систему счисления. Стек нужен для того, чтобы временно сохранить получаемые разряды числа́ и вывести их потом в порядке, обратном получению.

Пример 17.4. Перевод натурального числа в другую систему счисления с использованием стека

```
var x,a:integer;
{ Считаем, что здесь приведены описания подпрограмм
  Push, Pop и StackIsEmpty
  (которые мы опустили для лаконичности) }
begin
  write('Введите число и основание системы счисления:');
  readln(x,a);
  while x>0 do { Пока число не станет равным нулю }
  begin
    Push(x mod a); { Извлекаем из него младший разряд
                     и кладем его в стек }
    x:=x div a     { Вычеркиваем младший разряд числа }
  end;
  while not StackIsEmpty do { Пока стек не пуст }
    write(Pop)  { Извлекаем из стека цифры по одной
                  и выводим их на экран }
end.
```

Заметим, что данная программа работает только при условии, что основание не больше 10. В противном случае пришлось бы писать дополнительное условие для вывода на экран цифр, больших, чем 9.

Общий принцип применения стека — временное хранение цепочки данных для их последующего использования в обратном порядке.

При использовании стека очень важно перед каждым извлечением данных из стека проверять, не является ли он пустым.

ПРИМЕЧАНИЕ

Иногда возникает ситуация, когда из стека нужно извлечь верхний элемент, но извлеченное значение не нужно никуда использовать. В этом случае нужно просто использовать функцию `Pop` *так, как будто бы она является процедурой. То есть вместо того, чтобы писать* `write(Pop)` *или* `x:=Pop`*, можно просто написать отдельным оператором* `Pop;`*. Компилятор поймет, что нужно вызвать функцию, но ее значение не нужно никуда сохранять или выводить.*

ПРИМЕЧАНИЕ

В приведенных примерах считается, что в программе только один стек. Поэтому в подпрограммах используется глобальная переменная `pTop`*, которая неявно изменяется при использовании* `Push` *и* `Pop`*. Если возникает необходимость в использовании нескольких стеков, для каждой подпрограммы (*`Push`*,* `Pop` *и* `StackIsEmpty`*) нужно добавить дополнительный параметр-переменную — адрес вершины стека, передавать соответствующую переменную при каждом их вызове и использовать его при обращении к вершине стека.*

Задание 17.1. Напишите программу, которая вводит с клавиатуры числа, пока не будет введено число ноль. После этого программа выводит на экран введенные числа (кроме нуля) в обратном порядке.

Задание 17.2. Напишите программу, которая вводит с клавиатуры натуральное четное число *N*, после чего вводит последовательность из еще *N* целых чисел. Программа должна проверить, является ли введенная последовательность «зеркальной» (палиндромом). То есть если элементы последовательности переставить в обратном порядке, последовательность останется той же.

Задание 17.3. Напишите программу, которая вводит с клавиатуры числа, пока не будет введено четное число. После этого выводит на экран среднее в списке число (например, если ввели 7 или 8 чисел, нужно вывести четвертое по счету).

Задание 17.4. Напишите программу, которая вводит с клавиатуры числа, пока не будет введено число 100. Вывести на экран первую половину чисел, если их количество четно, либо вторую

(бо́льшую) половину чисел, если их количество нечетно. Числа выводить в том же порядке.

Программирование стека при помощи массива

Нам бы не хотелось, чтобы у читателя создалось впечатление, что стек — обязательно динамическая структура данных. Принцип LIFO очень удобен при программировании. В частности, для функционирования самого компьютера вызов любых подпрограмм и обработка любых событий (например, операций ввода-вывода) организованы при помощи аппаратно реализованного стека, при этом команды `Push` и `Pop` осуществляет сам процессор. Об этом вы можете прочитать в специальной литературе. Здесь мы собираемся только упомянуть, что стек можно реализовать при помощи обычного массива. Это даже удобнее, чем динамически, за исключением разве что обычной проблемы при использовании массивов — нужно заранее знать наибольшую длину массива (наибольшее число элементов стека). Нужно следить за тем, чтобы не произошло переполнение стека.

Пример 17.5. Реализация стека при помощи массива

```
const n=100; { Размер стека }
var Stack:array[1..n] of string; { Массив для хранения
                                   элементов стека }
    Top:integer=0; { Количество занятых ячеек стека.
                     Изначально все свободны }
procedure Push(chto:string);
begin
  if Top<n then { Если в стеке есть свободные ячейки }
  begin
    Top:=Top+1;      { Увеличиваем число занятых ячеек }
    Stack[Top]:=chto { Сохраняем новое значение }
  end
  else
  { Здесь хорошо бы как-то обрабатывать ситуацию
    переполнения стека. Можно, например, изменять
    значение некоей глобальной переменной, которую
    будет проверять вызывающая программа. Мы не будем
    приводить эту реализацию, чтобы не слишком
    усложнять программу }
end;
```

```
function Pop:string;
begin
  Pop:=Stack[Top]; { Извлекаем верхнее значение }
  Top:=Top-1       { Уменьшаем число занятых ячеек }
end;
function StackIsEmpty:boolean;
begin
  StackIsEmpty := Top=0
end;
```

Возможно, вы обратили внимание, что хотя для хранения стека используется совершенно другая структура данных, вызов подпрограмм ничем не отличается их вызовов в динамической реализации стека. Это очень удобно при программировании — если вы в какой-то момент решаете поменять внутреннее представление данных, достаточно будет только поменять тела подпрограмм, и ничего не менять в программе, которая их использует.

Выводы

1. Для выделения произвольного количества памяти во время работы программы нужно создать специальную структуру данных, состоящую из элементов, каждый из которых ссылается на другой, соседний элемент.
2. Самая простая такая структура данных называется стек. Для нее определены только две операции — положить элемент в стек и взять элемент из стека.
3. Операции помещения и изъятия элементов из стека определены только с одной стороны. В результате элемент, положенный в стек последним, будет изъят из него первым.
4. В простейшем случае последовательное помещение данных в стек и последующее их последовательное изъятие поменяет порядок следования данных на противоположный.

Контрольные вопросы

1. Какая основная идея того, как хранить в памяти множество элементов, заранее не зная их количества? Какая структура элемента данных для этого используется?

2. Как при описании типа данных для цепочки динамических элементов можно задать «ссылку элемента на свой собственный тип данных» (чтобы не возникло ошибки задания типа данных, описание которого еще не закончено)?
3. Дайте определение стека.
4. Какую аббревиатуру используют для описания основного свойства стека? Что она означает?
5. Зачем необходимо использование вспомогательной переменной при помещении элемента в стек? При изъятии?
6. Для фрагмента программы:

   ```
   Push(1);
   Push(2);
   Push(3);
   Push(Pop+Pop);
   Push(4);
   Push(Pop*Pop);
   write(Pop-Pop);
   ```

   определить, что будет результатом его работы (считая, что в поле `data` стека хранятся не строки, как в наших примерах, а целые числа)?

ТЕМА 18

Динамические структуры данных. Очередь

Если при извлечении данных из хранилища нужно получать данные в порядке, обратном их помещению, использование стека — идеальное решение. Но часто встречается ситуация, когда данные нужно извлекать в том же порядке, в котором их туда помещали. При этом хранилище исполняет роль временного «отстойника», в котором данные накапливаются до того момента, пока не придет пора извлечь их в том же порядке.

Урок 18.1. Принцип работы и описание типа данных

Проанализировав указанный порядок помещения и изъятия элементов, получаем, что он происходит по принципу «первым пришел — первым ушел». Это принято называть FIFO (First In — First Out) — *очередь* (queue).

Работа программистской очереди действительно похожа на очередь в столовой за булочками (рис. 18.1) — каждый пришедший встает в конец очереди, а булочки выдают в начале очереди. Заметим, что в отличие от человеческой очереди, где каждый человек знает того, кто стоит непосредственно перед ним, компьютерная очередь организована наоборот (в обратную сторону) — каждый элемент «знает», кто стоит в очереди непосредственно за ним. Это различие возникает оттого, что люди организуют очередь и поддерживают порядок в ней сами. А в компьютерной очереди этот порядок поддерживает «тот, кто выдает булочки». Это можно представить себе так: новый человек, который встает в очередь, пишет на спине последнего человека в очереди свое имя, после чего сам становится последним. А при извлечении человека из очереди ему дают булочки, считывают у него со спины имя следующего и подзывают того к окошку.

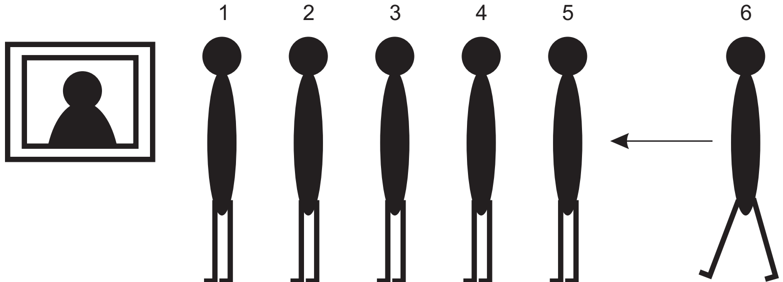


Рис. 18.1. Пример человеческой очереди

Получается, что структура каждого элемента очереди совпадает со структурой элементов в стеке — каждый хранит поле данных и ссылку на следующий элемент. Однако в отличие от стека, операции добавления и извлечения элементов из очереди происходят с разных сторон. Из начала элементы только удаляем, в конец только добавляем.

Значит, глобальных переменных, которые хранят информацию об очереди, требуется две: адрес первого и адрес последнего элемента очереди. Адрес первого элемента нужен, чтобы знать начало цепочки, какой элемент нужно извлекать из очереди первым (чья подошла очередь быть обработанным). А адрес последнего элемента нужно хранить, чтобы знать «того человека, на спину которого написали имя нового последнего», чтобы можно было легко добавить элемент в конец очереди.

Заметим, что для добавления элемента в конец не обязательно хранить адрес последнего элемента. Можно перебрать все элементы начиная с первого и последовательно добраться до последнего. Но эта операция может оказаться очень долгой при большой длине очереди. И тем более неэффективно совершать ее каждый раз при добавлении очередного элемента. Проще выделить одну дополнительную ячейку памяти и хранить адрес конца очереди.

Описание структуры данных очереди совпадает со структурой данных стека. Проще всего написать `type pQueue=pStack`. Если такое описание непонятно, приведем его полностью:

```
type pQueue=^Queue;
     Queue=record
        data:string;
```

```
    next:pQueue
  end;
```

Назовем требуемые глобальные переменные адресов начала и конца очереди pBegin и pEnd. Букву p в начало имени мы добавили, чтобы подчеркнуть, что это указатели и чтобы имена переменных не совпали со служебными словами begin и end.

```
var pBegin:pQueue=nil;
    pEnd  :pQueue=nil;
```

Так как очередь изначально пуста, мы присвоили обеим переменным указатели на nil.

Урок 18.2. Основные операции с очередью

Как мы уже говорили, при работе с очередью допустимы только две операции: поместить элемент в конец очереди и взять первый элемент из очереди. Опишем эти действия в виде подпрограмм.

Добавление элемента в очередь (EnQueue)

Назовем подпрограмму EnQueue (поместить в очередь). Для начала поймем, процедура это или функция. Результатом работы этой подпрограммы должно быть действие помещения указанного значения в очередь. Значит, никакого значения подпрограмма EnQueue возвращать не должна. То есть EnQueue — процедура. А так как для своей работы она должна «знать», что нужно положить в очередь, то это процедура с параметром. Так как параметр в результате работы процедуры изменяться не должен, передача параметра должна происходить по значению. То есть все как при помещении элемента в стек:

```
procedure EnQueue(chto:string);
```

chto — то значение, которое мы отправляем в очередь.

Процедура помещения элемента в очередь зависит от того, пуста очередь или в ней уже есть хотя бы один элемент. Различить эти два случая можно по указателю на начало очереди. Если в очереди нет ни одного элемента (указатель на начало очереди равен nil), нужно создать новый элемент (new(pEnd)), в его поле данных

положить chto (pEnd^.data:=chto), а его указатель на следующий элемент сослать на nil (pEnd^.next:=nil). После этого нужно указатель на начало очереди установить на этот же (единственный) элемент (pBegin:=pEnd).

Если в очереди есть хотя бы один элемент, нужно присоединить новый элемент к последнему элементу очереди: new(pEnd^.next). Заметим, что при этом происходит сразу два нужных нам действия: в памяти создается новый элемент, и последний элемент очереди сразу ссылается на него. Теперь остается переставить указатель конца очереди на этот новый элемент (pEnd:=pEnd^.next) и заполнить его поля. В поле данных положить chto (pEnd^.data:=chto), в поле ссылки на следующий элемент положить nil (pEnd^.next:=nil).

Сведем все вместе:

Пример 18.1. Процедура EnQueue (добавление элемента в конец очереди)

```
procedure EnQueue(chto:string);
begin
 if pBegin=nil then  { Если очередь пуста }
 begin
   new(pEnd);       { Выделяем память под новый элемент }
   pBegin:=pEnd;    { Указатель начала стека
                      устанавливаем на этот же элемент }
   pEnd^.data:=chto; { Записываем в него
                       сохраняемое значение }
   pEnd^.next:=nil { Указатель на следующий делаем nil }
 end
 else { Очередь не пуста }
 begin
   new(pEnd^.next);  { Выделяем память под новый элемент
                       и ссылаем последний элемент
                       очереди на него }
   pEnd:=pEnd^.next; { Перемещаем указатель конца
                       очереди на новый элемент }
   pEnd^.data:=chto; { Записываем в него
                       сохраняемое значение }
   pEnd^.next:=nil { Указатель на следующий делаем nil }
 end
end;
```

Возможно, вы заметили, что в обоих случаях последние два действия (заполнение полей вновь созданного элемента) совпадают. Поэтому можно записать данную процедуру немного короче:

Пример 18.2. Процедура EnQueue (добавление элемента в конец очереди). Более короткая запись

```
procedure EnQueue(chto:string);
begin
 if pBegin=nil then  { Если очередь пуста }
 begin
   new(pEnd);       { Выделяем память под новый элемент }
   pBegin:=pEnd     { Указатель начала стека
                      устанавливаем на этот же элемент }
 end
 else { Очередь не пуста }
 begin
   new(pEnd^.next);  { Выделяем память под новый элемент
                       и ссылаем последний элемент
                       очереди на него }
   pEnd:=pEnd^.next; { Перемещаем указатель конца
                       очереди на новый элемент }
 end;
 pEnd^.data:=chto; { Записываем в него
                     сохраняемое значение }
 pEnd^.next:=nil   { Указатель на следующий делаем nil }
end;
```

Извлечение элемента из очереди (DeQueue)

Назовем подпрограмму DeQueue (извлечь из очереди). По аналогии с извлечением элемента из стека приходим к выводу, что это функция без параметров, возвращающая строку:

```
function DeQueue:string;
```

В отличие от добавления элемента в очередь, при извлечении не обязательно рассматривать различные случаи. Даже случай единственного элемента в очереди, который на первый взгляд нужно рассматривать отдельно, вполне укладывается в общую схему. Сначала можно получить значение первого элемента очереди и выдать его в качестве значения функции (`DeQueue:=pBegin^.data`). После этого останется только переместить указатель начала

очереди на следующий элемент и очистить память, занимаемую первым элементом. Как и в случае со стеком, эти два действия не могут быть просто выполнены друг за другом. Нужно использовать вспомогательную переменную, чтобы, например, запомнить адрес той ячейки памяти, которую нужно будет удалить.

```
p:=pBegin;
DeQueue:=p^.data;
pBegin:=p^.next;
dispose(p);
```

Пример 18.3. Функция DeQueue (извлечение элемента из начала очереди)

```
function DeQueue:string;
var p : pQueue;   { Вспомогательная переменная }
begin
  p:=pBegin;      { Запоминаем адрес начала очереди }
  DeQueue:=p^.data; { Извлекаем данные из первого
                      элемента очереди }
  pBegin:=p^.next;{ Перемещаем указатель начала очереди
                    на следующий элемент }
  dispose(p)      { Освобождаем память }
end;
```

Проверка очереди на пустоту (QueueIsEmpty)

Как и в случае со стеком, нужна еще одна подпрограмма для работы с очередью — функция, которая проверяет очередь на пустоту (когда в очереди нет ни одного элемента). Это очень просто проверить — убедиться, что указатель на начало очереди равен `nil`.

Пример 18.4. Функция QueueIsEmpty (проверка очереди на пустоту)

```
function QueueIsEmpty:boolean;
begin
  QueueIsEmpty := pBegin=nil
end;
```

Урок 18.3. Использование очереди

Если профессионального программиста попросить привести пример использования очереди, он, прежде всего, скажет, что стек и очередь прекрасно используются при процедурах обхода дерева —

соответственно, для поиска в глубину или для поиска в ширину. Мы, к сожалению, не можем быть уверены, что вам на настоящий момент известно, о чем идет речь, поэтому приведем другой пример.

Рассмотрим решение такой задачи. На вход программе подается последовательность символов неизвестной длины, заканчивающаяся точкой. Других символов «точка» в последовательности нет. Необходимо вывести на экран слова, содержащиеся в последовательности, которые длиннее, чем самое короткое слово последовательности. Словом будем считать любую последовательность символов, отделенную от другого слова одним или несколькими пробелами. Известно, что каждое слово не длиннее 50 символов. Слова вывести по одному в строке, в том же порядке.

Очевидно, что входную последовательность необходимо просмотреть два раза — для определения самого короткого слова и для отбора только тех слов, которые длиннее его. Определить длину самого короткого слова в общем случае можно не раньше, чем будут перебраны все слова. Значит, нужно где-то хранить всю (в худшем случае) последовательность слов. Выделить под это массив нельзя, потому что неизвестна его длина. Так как слова нужно будет вывести в том же порядке, наиболее разумно использовать очередь.

Пример 18.5. Вывести по одному все слова входной последовательности, которые длиннее самого короткого слова

```
{ Считаем, что здесь приведены описание типа данных
  очереди, описания подпрограмм EnQueue, DeQueue и
  QueueIsEmpty, а также переменные pBegin и pEnd
  (которые мы опустили для лаконичности) }
var s:string[51];
    ch:char;
    min:integer;
begin
 s:='';   { Здесь накапливаем текущее слово }
 min:=51; { Минимальная длина слова }
 repeat
   read(ch); { Читаем очередной символ }
   if (ch<>' ')and(ch<>'.') then
              { Если не конец слова }
     s:=s+ch  { Добавляем символ к слову }
   else
```

```
    if s<>'' then { Если конец имеющегося слова }
    begin
      if length(s)<min then { Не являетя ли длина слова
                              минимальной }
        min:=length(s);
      EnQueue(s);  { Добавляем слово в очередь }
      s:=''        { "Обнуляем" слово }
    end
  until ch='.'; { Конец последовательности символов }
  while not QueueIsEmpty do
  begin
    s:=DeQueue;  { Извлекаем слово из очереди }
    if length(s)>min then  { Если оно длиннее min }
      writeln(s)           { Выводим его на экран }
  end
end.
```

Общий принцип применения очереди — временное хранение цепочки данных для их последующего использования в том же порядке.

ПРИМЕЧАНИЕ

Как и в случае со стеком, во всех приведенных примерах считается, что в программе только одна очередь. Поэтому в подпрограммах используются глобальные переменные pBegin *и* pEnd*, которые неявно изменяются при использовании* EnQueue *и* DeQueue*. Как и в случае с несколькими стеками, если возникает необходимость в использовании нескольких очередей, нужно для каждой подпрограммы (*EnQueue, DeQueue *и* QueueIsEmpty*) добавить дополнительные параметры-переменные — адрес начала очереди для* DeQueue *и* QueueIsEmpty *и адреса начала и конца очереди для* EnQueue *и передавать соответствующие переменные при каждом их вызове и использовать их при реализации всех трех подпрограмм.*

Задание 18.1. Напишите программу, которая вводит с клавиатуры числа, пока не будет введен ноль. После этого программа выводит на экран количество введенных чисел, а затем все эти числа в том же порядке.

Задание 18.2. Напишите программу, которая вводит с клавиатуры последовательность целых чисел, заканчивающуюся нулем. Программа должна проверить, выполняется ли для последовательности такое правило: если входную последовательность разделить

на положительные и отрицательные числа (записанные в том же порядке), то эти две последовательности будут совпадать (без учета знака, конечно).

Программирование очереди при помощи массива

Как и в случае со стеком, очередь — не обязательно динамическая структура данных. Принцип FIFO так же очень удобен при программировании, как и LIFO. В частности, при помощи очереди запрограммировано функционирование клавиатуры компьютера. При нажатии каждой клавиши ее код помещается в специальную очередь кодов нажатых клавиш, откуда процессор извлекает эти коды по порядку в тот момент, когда находит время для реакции на нажатия клавиш.

Здесь мы собираемся показать, как очередь можно реализовать при помощи обычного массива. Хотя это удобнее, чем динамически, напоминаем об обычной проблеме при использовании массивов — нужно заранее знать наибольшую длину массива (наибольшее число элементов очереди).

Заметим, что при реализации очереди в массиве возникает проблема эффективности. Если считать, что начало очереди — в первом элементе массива, то добавление элемента в конец очереди не вызывает проблем. Считаем, что позиция конца очереди хранится в целочисленной переменной, и увеличиваем ее при добавлении элемента.

Но если считать, что начало очереди находится в первом элементе массива, то при извлечении элемента из очереди возникает необходимость сдвинуть все элементы массива на одну позицию в сторону начала массива. Очевидно, что при значительной длине массива это весьма неэффективно. Хотелось бы, чтобы извлечение элемента из очереди было так же быстро, как и добавление элемента. Используем еще одну переменную — позицию начала очереди в массиве и будем считать, что очередь хранится в массиве между элементами, номера которых хранятся в этих целочисленных переменных. Назовем их для определенности `iBegin` и `iEnd` (индексы начала и конца очереди).

Еще одна тонкость, которую нужно предусмотреть: что делать, когда конец очереди находится в последнем элементе выделенного массива, и нужно добавить еще один элемент. Если при этом

сдвигать всю очередь в начало массива, снова получается неэффективно. Нужно считать, что массив «замкнут циклически», что после последнего элемента массива идет первый (то есть $n+1=1$). Заметим, что при этом индекс начального элемента очереди не всегда будет меньше, чем индекс последнего.

Еще один вопрос, который стоит задать: что будет признаком того, что очередь пуста или, наоборот, полна? Ведь оба индекса — `iBegin` и `iEnd` — могут принимать любые значения от `1` до `n`. На первый взгляд, можно считать очередь заполненной, если конец «догнал» начало, то есть `iEnd` «на 1 меньше», чем `iBegin`. «Меньше» в кавычках означает «с учетом циклической замкнутости очереди в массиве». Но этот же признак хочется считать пустой очередью. Чтобы отличать эти два состояния, нужно не допускать того, чтобы `iEnd` оказался «на 1 меньше» чем `iBegin`. Тогда заполненная очередь — когда `iEnd` «на 2 меньше» чем `iBegin`.

Пример 18.6. Реализация очереди при помощи массива

```
const n=100; { Размер очереди }
var Queue:array[1..n] of string; { Массив для хранения
                                   элементов очереди }
    iBegin:integer=1; { Индекс начального элемента
                        очереди в массиве }
    iEnd:integer=n;   { Индекс конечного элемента
                        очереди в массиве }
procedure EnQueue(chto:string);
begin
  { Если в очереди есть свободные ячейки }
  if (iBegin-iEnd+n) mod n<>2 then
  begin
    if iEnd<n then { Конец не в последнем элементе }
      iEnd:=iEnd+1
    else           { Конец в последнем элементе }
      iEnd:=1;     { Перемещаем конец в начало }
    Queue[iEnd]:=chto;
  end
  else
    { Здесь нужно как-то обрабатывать ситуацию
      переполнения очереди }
end;
function DeQueue:string;
begin
```

```
 DeQueue:=Queue[iBegin];
 if iBegin<n then { Начало не в последнем элементе }
   iBegin:=iBegin+1
 else            { Начало в последнем элементе }
   iBegin:=1     { Перемещаем начало в первый элемент }
end;
function QueueIsEmpty:boolean;
begin
  QueueIsEmpty := (iBegin-iEnd+n) mod n=1
end;
```

В этой программе мы использовали не совсем очевидный способ вычисления признаков «меньше на 1» и «меньше на 2».

Выводы

1. Чтобы извлечь из структуры данных элементов в том же порядке, используют очередь.
2. В очередь элементы можно добавлять только в конец, а извлекать — только из начала.
3. Для быстрого добавления и извлечения данных из очереди хранят адреса не только начала, но и конца очереди.

Контрольные вопросы

1. В чем отличие очереди людей от компьютерной очереди?
2. Каков критерий целесообразности применения очереди?
3. Почему далеко не всегда реализуют очередь при помощи массива?

ТЕМА 19

Динамические структуры данных. Однонаправленный список

Мы подробно рассмотрели структуры данных, в которых операции добавления и извлечения элементов строго ограничены правилами — FIFO и LIFO (очередь и стек). Теперь пришло время рассмотреть структуру данных, которая организована по точно такому же принципу, но в которой отсутствуют какие-либо ограничения на способ добавления/удаления элементов.

Урок 19.1. Описание типа данных и принцип работы

Динамическая структура данных, элементы которой хранятся последовательно, причем каждый элемент хранит в себе ссылку на следующий элемент, называется *однонаправленным списком*. Как и в случае со стеком и очередью, для указания конца однонаправленного списка в последнем элементе хранится ссылка на `nil`.

Чтобы иметь возможность обращаться к однонаправленному списку, достаточно хранить только одну переменную — адрес начала списка.

Как мы уже писали в теме 17, описание структуры данных для однонаправленного списка такое же, как у стека и у очереди:

```
type pListElement=^ListElement;
     ListElement=record
        data:string;
        next:pListElement
     end;
```

Глобальную переменную, хранящую адрес начала списка[1], назовем `pBegin`:

```
var pBegin:pListElement=nil;
```

---

[1] В этой теме мы будем использовать термин «список» как синоним термина «однонаправленный список». В общем случае список может быть каким-нибудь еще, например двунаправленным.

Можно представлять себе список так, как показано на рис. 19.1. Важно понимать, что при использовании стека и очереди операция получения значений элементов обязательно сопровождается удалением самих элементов из структуры данных. Для однонаправленного списка это вовсе не обязательно. Скорее наоборот, обычной ситуацией для списка является операция просмотра значений всех элементов и либо проверка их значений, либо осуществление с ними каких-либо действий. При этом удаления элементов из списка обычно не происходит.

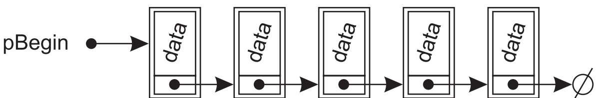


Рис. 19.1. Организация однонаправленного списка

Урок 19.2. Основные операции с однонаправленным списком

При работе со списком нужно уметь добавлять элемент в список, удалять из него элемент и последовательно просматривать все элементы списка.

Последовательный просмотр всех элементов списка

Назовем подпрограмму `PrintList` (печать списка). Чтобы перебрать все элементы списка, нужно ввести дополнительную переменную-указатель, которая будет по очереди «пробегать» адреса всех элементов списка, от первого до последнего. Функция этой переменной очень похожа на функцию счетчика цикла при переборе всех значений одномерного массива.

Так как количество элементов списка нам, в общем случае, неизвестно, будем использовать цикл `while`. Перед началом цикла присвоим переменной цикла значение адреса начала списка. Будем перемещать эту переменную на следующий элемент списка до тех пор, пока ее значение не станет равно `nil` (признаку конца списка).

Пример 19.1. Процедура PrintList (просмотр всех элементов однонаправленного списка)

```
procedure PrintList;
var p:pListElement; { Временная переменная-указатель
                      на элементы списка }
begin
 p:=pBegin;         { Устанавливаем ее начальное
                      значение на адрес начала списка }
 while p <> nil do  { Пока не достигнем конца списка }
 begin
   writeln(p^.data);{ Здесь можно осуществить любое
                      требуемое действие с текущим
                      элементом списка }
   p:=p^.next       { Перемещаем переменную-указатель
                      на следующий элемент списка }
 end
end;
```

Помещение элемента в список

Назовем подпрограмму PutInList (поместить в список). По аналогии с рассуждениями о добавлении элементов в стек и в очередь это процедура с параметрами. Но в отличие от очереди и стека, где помещение элементов происходит в строго определенное место, в списке это ограничение отсутствует. В общем случае нужно уметь помещать элемент в начало списка, в конец, в середину и в пустой список. Значит, нужно передавать этой процедуре параметр, указывающий место списка, в которое нужно поместить добавляемый элемент. Так как про значения элементов списка мы ничего не знаем, эта информация должна быть указателем (адресом). Рассмотрим случай помещения элемента в список — в середину (рис. 19.2).

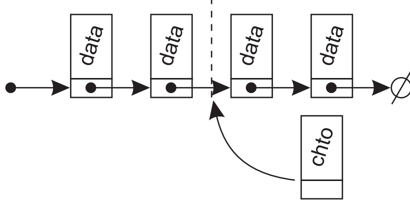


Рис. 19.2. Общий случай помещения нового элемента в однонаправленный список

При добавлении элемента в его поле next нужно поместить адрес элемента, который следует после него (это нетрудно). А адрес нового элемента нужно поместить в поле next того элемента, который будет находиться в списке непосредственно перед ним. Но для этого нужно знать адрес этого элемента (после которого мы помещаем новый элемент). Это и должно быть значением второго параметра процедуры PutInList.

```
procedure PutInList(chto:string;
                    kuda:pListElement);
```

chto — то значение, которое мы помещаем в список после элемента, адрес которого указан в параметре kuda.

Проанализировав действия, которые нужно совершить при помещении элемента в середину и в конец списка, можно обнаружить, что они совпадают. Поэтому можно считать эти два случая одним. Тот же вывод можно сделать, проанализировав случаи помещения элемента в начало списка и в пустой список. Получаем всего два случая: помещение в начало списка и после указанного элемента. Но при помещении в начало списка нельзя указать адрес элемента, после которого следует вставлять в список новый элемент. Значит, нужно условиться, как сообщить процедуре, что элемент следует поместить в начало. Проще всего в этом случае в качестве параметра kuda передать nil (не существует элемента, после которого нужно вставить новый).

Сведем все сказанное вместе:

Пример 19.2. Процедура PutInList (помещение элемента в однонаправленный список)

```
procedure PutInList(chto:string; kuda:pListElement);
var p:pListElement; { Временная переменная-указатель }
begin
 if kuda=nil then   { Помещаем элемент в начало списка }
 begin
   new(p);         { Выделяем память под новый элемент }
   p^.next:=pBegin;{ Отсылаем его на начало списка }
   pBegin:=p;      { Указатель начала стека
                     устанавливаем на этот элемент }
   p^.data:=chto   { Записываем в новый элемент
                     сохраняемое значение }
 end
 else { Помещаем элемент после указанного }
 begin
```

```
    new(p);         { Выделяем память под новый элемент }
    p^.next:=kuda^.next;  { Ссылку с нового элемента
                      на следующий делаем равной адресу
                      элемента, стоявшего после kuda }
    kuda^.next:=p;  { Устанавливаем ссылку
                      с элемента kuda на новый }
    p^.data:=chto   { Записываем в новый элемент
                        сохраняемое значение }
 end
end;
```

Возможно, вы заметили, что в обоих случаях первое и последнее действия совпадают. Более короткую версию процедуры PutInList предлагаем написать самостоятельно.

Удаление элемента из списка

Вы, конечно, обратили внимание, что из стека и очереди мы извлекали элементы, а из списка удаляем. Еще раз подчеркнем, в случае стека и очереди при получении хранимого значения оно тут же удаляется из структуры данных. Получение значений из списка и их удаление далеко не всегда совпадают, поэтому мы рассматриваем эти действия отдельно.

Так как удаляемое значение не требуется возвращать в вызывающую программу, подпрограмма RemoveFromList (удалить из списка) будет процедурой. Ей нужно указать параметр — какой элемент нужно удалять. Рассмотрим самый общий случай — удаление элемента из середины списка. Наиболее удобным представляется сообщить процедуре RemoveFromList непосредственный адрес удаляемого элемента.

Анализируем действия, которые при этом требуется совершить: установить ссылку элемента, стоящего перед удаляемым, на элемент, стоящий после удаляемого, и освободить память. Получить адрес элемента, стоящего после удаляемого, просто. Он хранится в поле next. А вот адрес предыдущего элемента получить проблематично. Ведь у нас список однонаправленный. В обратную сторону по такому списку не пройти. Единственное, что можно сделать — перебирать все элементы от начала, пока поле next не будет содержать адрес удаляемого элемента. Но это очень долго. Лучше придумать что-нибудь эффективнее. Проще передавать не адрес удаляемого элемента, а адрес предыдущего. Зная его, получить адрес удаляемого и адрес следующего очень легко.

По аналогии с добавлением элемента в список при удалении в общем случае существуют четыре случая: удаление элемента из середины списка, из конца, из начала и удаление единственного элемента. Но необходимые для этого действия, как и при добавлении, попарно совпадают.

Для указания того, что удаление элемента должно происходить из начала списка, в качестве адреса того элемента, за которым стоит удаляемый, будем передавать `nil`.

Пример 19.3. Функция RemoveFromList (удаление элемента из списка)

```
procedure RemoveFromList(gde:pListElement);
var p : pListElement;   { Вспомогательная переменная }
begin
 if gde = nil then  { Удаление первого элемента }
 begin
   p:=pBegin;       { Запоминаем адрес первого элемента }
   pBegin:=p^.next; { Переставляем начало списка
                      на следующий элемент }
   dispose(p)       { Освобождаем память, занимаемую
                      первым элементом }
 end
 else               { Удаление не первого элемента }
 begin
   p:=gde^.next;    { Запоминаем адрес
                      удаляемого элемента }
   gde^.next:=p^.next; { Переставляем ссылку
                         с предыдущего элемента на следующий }
   dispose(p)       { Освобождаем память, занимаемую
                      удаляемым элементом }
 end
end;
```

Заметим, что такая реализация процедуры `RemoveFromList` добавляет некоторые неудобства при программировании вызывающей программы, но зато эффективна по времени выполнения.

Урок 19.3. Обработка списков

Использование списков оправдано всегда, когда максимальный размер данных заранее неизвестен и при этом требуется хранить множество элементов, которое периодически нужно последовательно просматривать.

В качестве примера обработки списка приведем программу, которая вставляет в список новый элемент X перед каждым вхождением в список элемента Y. Заметим, что данная задача имеет смысл только при $X \neq Y$.

При составлении программы нужно обратить внимание на два момента. Во-первых, если просто перебирать элементы списка и сравнивать каждый из них с Y, возникает проблема добавления в список элемента перед найденным. Ведь ссылку на новый элемент нужно поместить в поле next элемента, стоящего перед Y, а мы его уже «прошли». Значит, нужно проверять на равенство Y не текущий элемент списка, а следующий после текущего. Тогда нам будет известен адрес элемента, в котором нужно изменить поле next. Во-вторых, нужно отдельно рассмотреть ситуацию, если первый элемент в списке равен Y. Ведь перед первым элементом нет предыдущего, поэтому под рассмотренную общую схему он не попадает.

Пример 19.4. Добавление в однонаправленный список нового элемента, равного X, перед каждым вхождением в список элемента Y

```
var x,y:string;
{ Считаем, что здесь приведено описание процедуры
  PutInList (которое мы опустили для лаконичности) }
    p:pListElement; { Временная переменная }
begin
  write('Введите X:');
  readln(x);
  write('Введите Y:');
  readln(y);
  p:=pBegin; { Устанавливаем "p" на начало списка }
  if p = nil then { Для пустого списка задача
                    не имеет смысла }
  begin
    writeln('Список пуст');
    exit
  end;
  if p^.data = y then { Если X в начале списка }
    PutInList(y,nil); { Добавляем перед ним Y }
  while p^.next <> nil do { Пока не достигнем
                            конца списка }
  begin
    if p^.next^.data = x then { Если элемент,
```

```
                    следующий после текущего,
                    равен X }
    begin
      PutInList(y,p); { Вставляем перед ним Y }
      p:=p^.next      { Переходим к следующему }
    end;
    p:=p^.next        { Переходим к следующему }
  end
end.
```

Заметим, что в случае нахождения элемента, равного *X*, программа два раза выполняет переход к следующему элементу. Если этого не сделать, при простом переходе к следующему элементу текущим элементом станет *Y*, а следующим за ним снова будет *X*. Тогда перед ним снова будет вставляться *Y*, и так бесконечно.

ПРИМЕЧАНИЕ

Как и в случае со стеком и очередью, во всех приведенных примерах считается, что в программе только один список. При использовании нескольких списков не забудьте описать нужное количество переменных начала списка, добавить параметр-указатель в каждую процедуру и использовать его при обращении к списку.

Целесообразность использования однонаправленного списка

Вы, конечно, понимаете, что однонаправленный список является альтернативой использования одномерного массива. Нам бы хотелось, чтобы вы научились правильно выбирать эффективную структуру данных для каждой задачи.

Таблица 19.1. Сравнение эффективности работы с одномерным массивом и с однонаправленным списком

Операция	Одномерный массив	Однонаправленный список
Последовательный просмотр всех элементов	Одинаковая сложность	
Обращение к произвольному элементу	Прямой доступ. Очень быстро	Последовательный перебор элементов. Долго

Операция	Одномерный массив	Однонаправленный список
Добавление (вставка) или удаление элемента	Сдвиг всех следующих элементов. Долго	Небольшое число операций. Быстро
Обращение к одному из соседних элементов, в произвольном направлении	Прямой доступ. Очень быстро	Последовательный просмотр, в худшем случае большинства элементов. Долго
Выделение памяти	Нужно знать максимальное число элементов в момент разработки программы	Можно выделить память в произвольный момент работы программы

Изучив приведенную таблицу, можно сделать вывод, что однонаправленный список эффективно использовать при частой вставке/добавлении элементов, при неизвестном заранее ограничении числа элементов и при отсутствии необходимости обращаться к произвольному элементу списка. Одномерный массив эффективно использовать при заранее известном ограничении на число элементов, отсутствии необходимости вставлять/удалять элементы и при необходимости доступа к произвольному элементу.

Задание 19.1. Напишите программу, которая создает однонаправленный список и заполняет его случайным количеством (10–20) случайных целых чисел. Вывести список на экран. Найти среднее арифметическое всех четных элементов списка.

Задание 19.2. Для списка задания 19.1 напишите подпрограмму, которая:

а) разворачивает список в обратном порядке за один проход по списку;

б) подсчитывает число элементов списка, которые больше предыдущего, но меньше следующего;

в) проверяет, есть ли в списке хотя бы два одинаковых элемента;

г) из каждой группы подряд идущих равных элементов списка оставляет только один;

д) порождает из списков A и B список C, который состоит из элементов, которые содержатся только в A или только в B.

Выводы

1. Самая простая динамическая структура данных, не имеющая ограничений о месте добавления/удаления элементов — однонаправленный список.
2. Каждый элемент однонаправленного списка содержит в себе адрес следующего элемента.
3. Для хранения однонаправленного списка достаточно использовать только одну глобальную переменную — адрес первого элемента.
4. Основные действия по обработке однонаправленного списка — последовательный просмотр всех элементов, добавление и удаление элемента в указанном месте.

Контрольные вопросы

1. Какой тип данных может храниться в поле `data` однонаправленного списка?
2. Какое начальное значение имеет переменная-указатель начала списка?
3. Какую алгоритмическую конструкцию имеет смысл использовать для просмотра всех элементов списка?
4. При просмотре массива для перехода к следующему элементу используется увеличение счетчика цикла. Какой аналог этой операции используется для перехода к следующему значению списка?
5. Какой дополнительный параметр приходится передавать в процедуры добавления и удаления элемента списка? Чем неудобен этот параметр, особенно при удалении элемента? Почему все же применяют именно такое решение?
6. Какой отдельный случай необходимо рассматривать при добавлении/удалении элементов? Как вызывающая программа должна обозначить этот случай?

ТЕМА 20
Рекурсия

Решение ряда задач (не обязательно алгоритмических) оказывается простым и удобным, если считать, что умеешь решать такую же задачу, но меньшего размера. Такое соотношение между задачей большей и меньшей размерности называется *рекуррентным*, а метод решения — *рекурсией*.

Урок 20.1. Описание принципа

Программа, которая в процессе работы вызывает саму себя, но меньшей размерности, называется *рекуррентной*. В Паскале нельзя сделать рекуррентной саму программу, но можно написать рекуррентную подпрограмму (процедуру или функцию). Такая подпрограмма будет содержать в своем теле хотя бы один вызов подпрограммы с точно таким же именем[1].

Хотелось бы сразу предупредить вас, что для самостоятельного написания рекуррентной подпрограммы требуется особенный подход в мышлении. По крайней мере нам не известно гарантированной методики обучения рекуррентному программированию. Мы постараемся на примерах показать вам основные принципы рекурсии и программирования рекуррентных подпрограмм. Надеемся, что это позволит вам проникнуться красотой и очарованием рекуррентного решения задач и научиться их реализовывать.

[1] Вообще говоря, не обязательно, чтобы подпрограмма вызывала саму себя. Она может вызывать другую подпрограмму, которая будет в свою очередь вызывать первую. Или эта конструкция может быть еще более сложной и запутанной. Во всех этих случаях принято говорить о рекурсии.

Надеемся, вы обратили внимание на главный признак («закон») рекурсии — использование решения такой же задачи, но меньшей размерности. Решение большой задачи ссылается на решение (такое же) меньшей. То, в свою очередь, еще меньшей. И так до... до каких пор? Здесь возникает второй «закон» рекурсии: для некоторой размерности, самой маленькой, решение должно быть примитивным и известным. При ссылках на решения все более мелких задач нужно когда-нибудь остановиться.

В качестве первого примера приведем определение факториала. Нерекуррентное определение: $n!$ — произведение первых n натуральных чисел.

Рекуррентное определение состоит из двух частей. Первая часть — рекуррентный переход: $n! = (n - 1)! \times n$. Вторая часть — остановка: 1! = 1. Так, при рекуррентном вычислении 4! вызовется программа вычисления 3!, она вызовет вычисление 2!, она вызовет вычисление 1!, его значение известно (=1) и будет выдано для вычисления 2! (=1 × 2), его результат (=2) — для вычисления 3! (=2 × 3), а его результат (=6) — для вычисления, наконец, 4! (=6 × 4=24). Этот процесс отражен на рис. 20.1. При каждом вызове рекурсии процесс вычисления приостанавливался, запоминая свое текущее состояние, и происходило вычисление такой же задачи, но меньшей размерности.

```
4!=3!*4                 =6*4=24
  └→3!=2!*3              =2*3=6
        └→2!=1!*2        =1*2=2
              └→ 1!=1 ┘
```

Рис. 20.1. Последовательное обращение к подпрограмме при рекуррентном вычислении факториала

Рассмотрим подпрограмму на Паскале, которая работает описанным образом. Так как результатом подпрограммы является число, используем функцию, возвращающую целочисленное значение[1].

[1] Значения функции факториал $n!$ даже при не очень больших значениях n становятся очень большими. Поэтому правильнее было бы использовать для типа возвращаемого значения функции большую размерность, чем `integer`. Хотя бы `longint`.

Пример 20.1. Рекуррентная функция вычисления факториала

```
function Fact(n:integer):integer;
begin
 if n=1 then          { Если вычисляется 1!,
                        его значение известно }
   Fact:=1            { и равно 1 }
 else                 { Если n не равно 1 }
   Fact:=Fact(n-1)*n  { Используем рекуррентный вызов }
end;
```

Возможно, вы уже оценили элегантность такого решения по сравнению с традиционным (которое использует цикл от 1 до *n* и накапливает значение факториала последовательно). Но очень важно сразу предупредить вас, что использовать рекуррентное решение нужно только при необходимости. Проблема в том, что при реализации рекурсии происходит вызов подпрограмм. А вызов подпрограмм для компьютера сопряжен с дополнительными накладными расходами[1]. Поэтому если нерекуррентное решение известно и не требует от программиста значительных усилий на свою реализацию, предпочтительнее использовать его. Использование рекурсии целесообразно только при значительных затруднениях в программировании другого, нерекуррентного, варианта. Либо при его отсутствии.

Задание 20.1. Напишите рекуррентную программу, которая вводит с клавиатуры число *n* и вычисляет *n*-й элемент последовательности Фибоначчи (последовательность чисел, начинающаяся с двух единиц, в которой каждый последующий элемент является суммой двух предыдущих).

ПРИМЕЧАНИЕ

Мы просим вас вычислить элемент последовательности Фибоначчи рекуррентным способом только для того, чтобы вы на этом простом

---

[1] Если вам интересно, при вызове подпрограмм процессор должен приостановить вычисление текущей задачи и начать решать новую. Чтобы он имел возможность вернуться в то же свое состояние, в котором находился до приостановки, процессор должен запомнить всю текущую «среду вычислений». А после возврата в это место — восстановить всю «среду вычислений» обратно. Это запоминание и восстановление требует некоторого, не слишком короткого, времени.

и понятном примере потренировались писать рекуррентные программы. В действительности числа Фибоначчи ни в коем случае не нужно вычислять рекуррентно, потому что при этом множество начальных значений последовательности вычисляется заново много раз. При этом сложность рекуррентного решения составляет примерно 2^n (где n — номер вычисляемого элемента последовательности). В то время как последовательное вычисление всех элементов по порядку, запоминая всего лишь два предыдущих, требует порядка 3×n действий.

Урок 20.2. Ханойские башни

Одним из лучших примеров целесообразности использования рекурсии является решение задачи о Ханойских башнях. Постановка задачи такова: имеется три стержня, на один из которых надета стопка из нескольких дисков (n), все они разного размера, снизу большего диаметра, сверху — меньшего (как детская пирамидка, рис. 20.2). Нужно переложить все диски с этого стержня на соседний. При этом необходимо соблюдать два правила: 1) перекладывать диски можно только по одному; 2) никогда диск большего диаметра не должен лежать поверх диска меньшего диаметра.

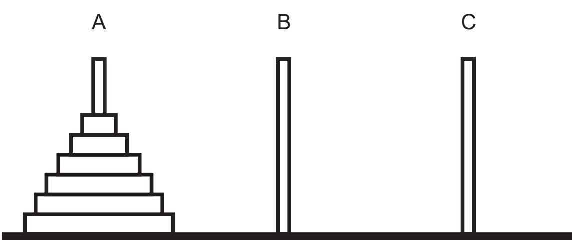


Рис. 20.2. Ханойские башни

Для небольшого количества дисков (для трех, например) задача кажется несложной. Но при увеличении числа дисков сложность нахождения решения (последовательности перекладывания дисков) резко увеличивается. Нахождение универсального решения представляется затруднительным.

Проанализировав решение, можно увидеть, например, следующее. Чтобы переложить стопку с одного стержня на другой,

нужно в какой-то момент переложить самый нижний диск. Это возможно только в том случае, если на первом стержне надето это самое большое кольцо, а второй стержень пустой. Значит, все остальные диски ($n-1$) находятся на третьем диске. Получается, нужно переложить стопку из $n-1$ дисков на третий стержень, потом переложить последний диск на второй стержень. Остается переложить стопку из $n-1$ диска на второй стержень. Такое описание решения как раз и является рекуррентным. Запишем его более формально.

Пример 20.2. Рекуррентное описание решения задачи о Ханойских башнях

```
procedure ПереложитьСтопкуДисков(
            N { Сколько дисков }        ,
            A { С какого стержня }      ,
            B { На какой стержень }     ,
            C { Через какой стержень } :integer);
begin
 if N = 1 then
   ПереложитьДиск(A, B)
 else
 begin
   ПереложитьСтопкуДисков(N-1, A, C, B);
   ПереложитьДиск(A, B);
   ПереложитьСтопкуДисков(N-1, C, B, A)
 end
end;
```

Эта процедура, конечно, написана не на Паскале. Мы сохранили структуру процедуры Паскаля, но позволили себе назвать процедуру по-русски и использовали вызов несуществующей процедуры `ПереложитьДиск`. Также мы сделали допущение, что стержни пронумерованы целыми числами. Нам было важно прояснить смысл без ущерба для понимания, но при этом мы постарались максимально приблизить процедуру к синтаксису Паскаля.

Задание 20.2. Исправьте приведенную в примере 20.2 процедуру так, чтобы она выводила на экран последовательность перекладывания дисков, необходимую для перекладывания стопки из 5 дисков. Убедитесь, что результат содержит 31 перекладывание.

Урок 20.3. Структура рекуррентной подпрограммы

Пришло время сформулировать правила написания рекуррентной подпрограммы. Во-первых, обязательно нужно поставить условие остановки — условие того, что размерность задачи стала такой маленькой, что ее можно вычислить/выполнить простым, нерекуррентным способом. Во-вторых, организовать рекуррентный переход к задаче меньшей размерности. То есть вставить в тело подпрограммы вызов подпрограммы с таким же именем, что и сама описываемая подпрограмма, у которой меньший параметр размерности.

Самая простая рекуррентная программа — так называемая *хвостовая рекурсия*. В этом случае рекуррентный вызов стоит последней командой рекуррентной подпрограммы. Такое использование рекурсии легко для понимания, и мы ниже предложим вам написать несколько таких программ. Но нужно сознавать, что практического применения такое решение не имеет, потому что его можно легко превратить в цикл `while`, который, в отличие от многочисленных вызовов рекуррентных подпрограмм, не имеет накладных расходов при выполнении.

Для реализации хвостовой рекурсии можно пользоваться простой схемой:

1) проверяем условия выхода из рекурсии («начальное» значение аргумента);
2) если оно выполняется — присваиваем значение (и конец);
3) если нет — выполняем нужные на этом шаге действия и
4) вызываем себя с меньшим аргументом.

Пример 20.3. Рекуррентное решение задачи: с клавиатуры вводятся целые числа, пока не будет введен ноль; посчитать их сумму

```
function Sum:integer;
var x:integer;
begin
 readln(x);
 if x=0 then
   Sum:=0
 else
   Sum:=x+Sum
end;
```

Задание 20.3. Напишите рекуррентную программу, которая вводит с клавиатуры цифры, пока не будет введен 0, и выводит их в обратном порядке.

ПРИМЕЧАНИЕ

Использовать хвостовую рекурсию имеет смысл только для начального обучения написанию рекуррентных программ. При решении действительно рекуррентных задач хвостовой рекурсией не обойтись. Как правило, они содержат в себе несколько рекуррентных вызовов.

Урок 20.4. Пример рекуррентного решения нерекуррентной задачи

Рассмотрим задачу нахождения приближенного значения корня некоторой функции $f(x)$ на отрезке $[a,b]$. Решим эту задачу методом деления отрезка пополам. Решение действительно при условии, что на отрезке $[a,b]$ функция $f(x)$ непрерывна и имеет единственный корень. Будем искать решение с точностью ε, то есть найденный корень будет отличаться от истинного не более чем на ε.

Метод решения прост. Найдем середину отрезка и вычислим значение $f(x)$ в этой точке. Если значение $f(x)$ для левой границы отрезка имеет знак отличный от знака середины, то корень находится слева от середины. Если знаки одинаковы, то корень находится справа. Будем повторять это действие для соответствующей половины отрезка до тех пор, пока его длина не станет меньше ε. Тогда любую точку на отрезке можно считать приближенным значением корня.

В описании решения явно прослеживается рекуррентная схема: решение задачи сводится к решению такой же, но меньшей размерности (половина отрезка) и есть условие остановки рекурсии (длина отрезка меньше ε).

Пример 20.3. Рекуррентное решение задачи нахождения приближенного значения корня функции (методом деления отрезка пополам)

```
function Dichotomy(a,b,eps:real):real;
var c:real;
begin
```

```
    c:=(a+b)/2;
    if b-a<eps then
      Dichotomy:=c
    else
      if f(a)*f(c)<0 then
        Dichotomy:=Dichotomy(a,c,eps)
      else
        Dichotomy:=Dichotomy(c,b,eps)
  end;
```

При этом мы полагаем, что описание функции $f(x)$ известно и приведено ранее.

Хотя это решение данной задачи хорошо укладывается в рекуррентную схему, гораздо лучше использовать цикл `while` и обойтись парой дополнительных переменных a и b. Применение здесь рекурсии нецелесообразно.

Урок 20.5. Пример рекуррентного решения рекуррентной задачи

Пусть требуется вывести на экран все двоичные вектора веса n длины k, то есть все последовательности, состоящие из n цифр 0 и 1, в которых ровно k единиц. «Лобовое» решение этой задачи состоит в том, чтобы перебирать все возможные двоичные числа длины n, считать в них количество единиц и выводить на экран только те, в которых оказалось k единиц. Такое решение работает, но очень неэффективно. Очень большое количество чисел будет порождено зря и отброшено.

Правильнее научиться сразу порождать только нужные числа. При использовании рекурсии это оказывается легко и удобно. Алгоритм такой: поставим на первую позицию сначала 0, потом 1. Если мы поставили 0, то на оставшейся $n-1$ позиции нужно расставить все те же k единиц. А если мы поставили 1, то на оставшейся $n-1$ позиции осталось расставить еще $k-1$ единицу. Вот и все решение. Нужно только в каждом случае добавить проверку того, что такая расстановка возможна: что число расставляемых единиц не больше числа оставшихся позиций и что оно не меньше нуля. А также условие окончания рекурсии: что число позиций не стало равно 1 (тогда можно поставить на нее 0 или 1).

Осталось подумать еще об одном: как выводить получившуюся последовательность. Ведь наш алгоритм порождает последовательности постепенно, по одной цифре. При этом считается, что на текущую позицию ставим 0 или 1, а на тех позициях, которые уже рассмотрены, цифры уже расставлены. Значит, нужно где-то хранить эти расставленные цифры и выводить текущую последовательность в тот момент, когда ставится последняя цифра. Лучше всего для этой цели подходит глобальный массив длины большей, чем n. Чтобы при установке в массив новой цифры не высчитывать позицию, на которую она должна быть поставлена, проще расставлять цифры не с начала в конец, а с конца в начало. Тогда параметр n (число позиций, на которые предстоит расставить k единиц) и будет номером той позиции, на которую устанавливаем текущую цифру.

Последняя тонкость — рекуррентная подпрограмма должна знать исходную длину последовательности. Это нужно, чтобы знать, какой длины двоичную последовательность выводить на экран. Проще всего хранить ее в глобальной переменной.

Пример 20.4. Порождение всех двоичных последовательностей длины n, в которых ровно k единиц

```
var a:array[1..100] of integer; { Массив для хранения
                          порождаемой последовательности }
    i,dlina:integer; { Счетчик цикла и длина
                       последовательности
                       для вывода на экран }
procedure Rasstanovka(n,k:integer);
begin
 if n=1 then { Если осталась одна позиция }
 begin
   a[n]:=k;  { Устанавливаем на первую позицию 0 или 1.
               То есть число оставшихся единиц,
               (которое хранится в переменной k) }
   for i:=1 to dlina do
     write(a[i]);  { Выводим по одной все цифры }
   writeln
 end
 else
 begin
   a[n]:=0;  { Ставим на n-ю позицию "0" }
```

```
      if n-1>=k then  { Если число позиций не меньше
                        числа расставляемых единиц }
        Rasstanovka(n-1,k); { Расставляем
                              на n-1 позиции k единиц }
      a[n]:=1;  { Ставим на n-ю позицию "1" }
      if k-1>=0 then  { Если число расставляемых единиц
                        не меньше нуля }
        Rasstanovka(n-1,k-1); { Расставляем
                              на n-1 позиции k-1 единицу }
   end
  end;
  begin
    dlina:=6;
    Rasstanovka(dlina,3);
  end.
```

Выводы

1. Программа, которая в процессе своей работы обращается к самой себе, называется рекуррентной.
2. Рекуррентная программа должна содержать проверку условия окончания рекурсии и вызов самой себя меньшей размерности.
3. Использование рекурсии рационально только в том случае, если нет возможности написать нерекуррентный алгоритм.

Контрольные вопросы

1. Какими двумя свойствами должна обладать рекуррентная программа?
2. Чем хорошо и чем плохо использование рекуррентных подпрограмм?
3. В каком случае рекурсия называется хвостовой?

ПРИЛОЖЕНИЕ 1
Элементы блок-схем

Таблица П.1. Элементы блок-схем

Наименование	**Обозначение**	**Описание**
Начало/конец		Начало и конец любой блок-схемы
Действие		Любое простое (линейное) действие, кроме ввода-вывода, например оператор присваивания (`A := B + C`)
Условие		Ветвление алгоритма. Имеет два выхода — «Да» и «Нет». Проверяется условие, записанное внутри блока. Если условие истинно — далее выполняется часть блок-схемы по метке «Да». Если ложно — по метке «Нет»
Ввод-вывод		Общение алгоритма с внешним миром. Первое слово внутри блока всегда — «Ввод» или «Вывод». Первое слово означает, что данные из внешнего мира попадают в алгоритм (ввод); второе из алгоритма выдаются во внешний мир (вывод)
Процедура		Вызов алгоритма, написанного отдельным модулем (то есть имеющим свое «Начало» и «Конец»). Обычно это вспомогательный алгоритм

Наименование	Обозначение	Описание
Цикл со счетчиком	(шестиугольник)	Внутри блока обычно указывается выражение вида I = 1, N. Это означает, что нижеследующие блоки алгоритма (заканчивающиеся возвратом к блоку «Цикл со счетчиком») будут выполняться N раз. То есть на каждом шаге цикла переменная I будет последовательно принимать значения от 1 до N с шагом 1

Несколько примечаний к таблице.

1. На языке блок-схем оператор присваивания принято обозначать стрелкой влево — A ← B + C. Мы используем обозначение присваивания из языка Паскаль, так как в этой книге нам оно кажется более уместным.
2. На языке блок-схем можно установить шаг цикла, не равный единице: I = 3, N, 2 — переменная I меняется от 3 до N с шагом 2.
3. Иногда в литературе обозначают параметры изменения цикла со счетчиком так: $i \leftarrow \overline{1, N}$.
4. Все блоки соединяются между собой тонкими линиями, которые должны быть горизонтальными или вертикальными (но не наклонными). Соединительная линия всегда должна входить в блок сверху посередине, а выходить снизу посередине. Исключение составляет блок «условие», у которого выхода два и они могут отходить влево, вправо или вниз.
5. Каждый блок должен иметь ровно один вход (сверху) и ровно один выход (снизу). Исключение составляют блоки начала-конца, блок условия и цикл со счетчиком.

ПРИЛОЖЕНИЕ 2
Задачи

Integer. Описание. Ввод. Вывод. Операции

1. Введите с клавиатуры целое число. Выведите на экран следующее число, удвоенное значение числа, противоположное число, модуль числа, квадрат числа.
2. Введите с клавиатуры два целых числа. Выведите на экран сумму чисел, разницу между числами, квадрат разности между числами, частное и остаток от деления первого числа на второе.
3. Введите с клавиатуры два натуральных числа — длины сторон прямоугольника. Выведите на экран периметр и площадь прямоугольника.
4. Введите с клавиатуры координаты точки на плоскости (целочисленные). Выведите на экран квадрат расстояния от начала координат до этой точки.
5. Введите с клавиатуры целое число. Выведите на экран 4-ю, 6-ю и 8-ю степени числа. При вычислениях используйте минимальное количество действий.

Real. Описание. Ввод. Вывод. Операции и функции

6. Введите с клавиатуры вещественное число. Выведите на экран (в неэкспоненциальном виде) модуль числа, квадрат числа, умноженное на π число, арктангенс и тангенс числа.
7. Введите с клавиатуры вещественное число (радиус окружности). Выведите на экран длину этой окружности и площадь круга, который она описывает.

8. Введите с клавиатуры координаты точки на плоскости (два вещественных числа). Выведите на экран полярные координаты точки (расстояние от точки до начала координат, угол наклона прямой, проведенной в точку из начала координат, к прямой OX), синус, косинус и тангенс угла наклона.
9. Введите с клавиатуры вещественное число. Выведите на экран ее антье (целую часть числа), мантиссу (дробную часть числа) и ближайшее целое к этому числу.
10. Введите с клавиатуры два вещественных числа. Выведите на экран их сумму, разность, произведение, частное и первое число в степени второго.

Real. Запись и вычисление выражений

11. Записать на языке Паскаль:

а) $y = \left| x^2 + \dfrac{x + 2,5}{3x} \right| - 3\sqrt{\sin 2x - \dfrac{2}{1 - x}}$;

б) $y = 5\sqrt{\dfrac{2x - 3,5}{8x^2}} + e^{x+2} \cdot \cos 3x$;

в) $y = \dfrac{|5x - 6|}{\sqrt{x^2 - 3} + 2,4} - 3\ln\dfrac{2}{5x}$.

12. Записать на языке Паскаль:

а) $y = 2|x - 3|^{3x+5}$;

б) $y = \dfrac{5}{4\log_{x-2}|3 + x|}$;

в) $y = \dfrac{2tg^2 2x - 1}{4}$.

13. Вычислить по действиям значение выражения с указанием для каждого действия типа данных результата (Real или Integer):

а) `2-13 mod 7/3+sqr(4);`

б) `abs(8-sqrt(2/0.5)+4 mod 3);`

в) `12*3 div 5/3.5+sqr(1).`

Char. Описание. Ввод. Вывод. Функции[1]

14. Введите с клавиатуры символ. Выведите на экран три символа, следующие в таблице ASCII после введенного символа.
15. С клавиатуры вводятся две латинские буквы (неизвестно, больших или малых). Выведите на экран, стоит ли в алфавите первая буква перед второй (Да/Нет).
16. Введите с клавиатуры три строки. Выведите на экран последовательность четырех символов, составленную из: первого символа первой строки; второго символа второй строки; символа, следующего (в ASCII) после первого символа третьей строки; символа, предшествующего (в ASCII) второму символу третьей строки.
17. Введите с клавиатуры последовательность из четырех заглавных букв латинского алфавита. Проверьте, являются ли эти четыре символа подряд идущими буквами алфавита.
18. С клавиатуры вводятся три символа: цифра, символ плюс и еще одна цифра. Выведите на экран результат операции.

Boolean. Запись выражений

19. Известно, что в театральном буфете в продаже остались только шоколадки и шоколадные конфеты. И те и другие бывают молочные и черные (горькие), с орехами и без, с фруктовой начинкой и без начинки.

    Составьте для каждого пункта логическое выражение, которое истинно, если сладость, купленная в буфете является:

    а) молочной шоколадкой;

    б) конфетой черного шоколада с орехами;

    в) черной шоколадкой без начинки;

    г) изделием из молочного шоколада с начинкой без орехов;

    д) шоколадкой с начинкой или конфетой без орехов;

    е) изделием из черного шоколада либо с орехами, либо с начинкой (но не одновременно);

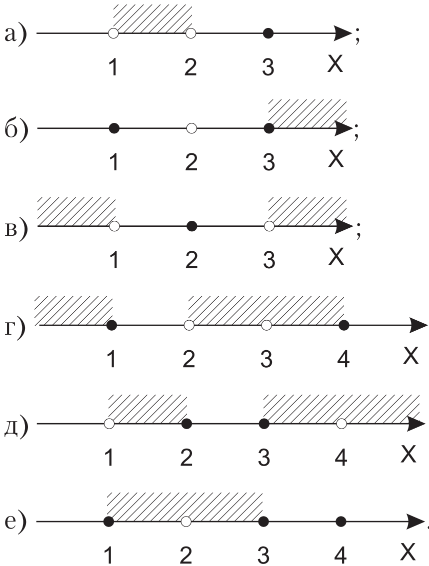
---

[1] При необходимости используйте функцию `upcase(x)`, она преобразует строчную латинскую букву в прописную. Любой другой символ остается без изменений.

ж) молочной конфетой без орехов и начинки или черной шоколадкой с орехами;
з) молочной шоколадкой без начинки или с орехами.
Результат каждого логического выражения выведите на экран.

Boolean. Вычисление выражений

20. Запишите на языке Паскаль выражение, которое истинно, если переменная x принадлежит заштрихованной области:



21. Запишите на языке Паскаль выражение, которое истинно, если переменная x:

а) принадлежит области $[-1; 1] \cup [3; 5]$;
б) не принадлежит области $[-1; 1] \cup [3; 5]$;
в) принадлежит области $[-\infty; 1] \cup [2; 4] \cup [5; +\infty]$.

If. Простые сравнения. Min/max/средний

22. Ввести с клавиатуры 2 различных числа. Вывести большее из них.

23. Ввести с клавиатуры 3 различных числа. Вывести меньшее из них.
24. Ввести с клавиатуры 3 различных числа. Вывести их в порядке возрастания.
25. Ввести с клавиатуры число. Вывести на экран знак числа (+1 — если число больше нуля, –1 — если число меньше нуля, 0 — если число равно нулю).
26. Ввести с клавиатуры 4 числа. Вывести на экран номер наибольшего из них.
27. Ввести с клавиатуры натуральное число. Вывести, является ли число двузначным.

If. Уравнения и неравенства с параметрами

Для заданий 28–32: для любых a и b, введенных с клавиатуры:

28. Решить уравнение $ax + b = 0$.
29. Решить уравнение $ax^2 + bx = 0$.
30. Решить неравенство $ax + b \leqslant 0$.
31. Решить уравнение $a|x| = b$.
32. Решить неравенство $ax^2 + b \geqslant 0$.
33. Решить уравнение $ax^4 + bx^2 + c = 0$ для любых a, b и c, введенных с клавиатуры.

For. Перечисления

34. Введите с клавиатуры 10 чисел. Выведите на экран произведение этих чисел.
35. Выведите на экран последовательность первых 20-ти нечетных чисел (1, 3, 5, 7, ...).
36. Выведите на экран первые 15 чисел последовательности: 1, 4, 7, 10, 13, ...
37. Выведите на экран последовательность четных чисел от –30 до +30.

38. Введите с клавиатуры два целых нечетных числа (первое больше второго). Выведите на экран последовательность нечетных чисел, расположенных между этими числами.
39. Выведите на экран первые 20 чисел последовательности Фибоначчи: 1, 1, 2, 3, 5, 8, ... (каждый последующий равен сумме двух предыдущих).

For. Вычисления со счетчиком цикла

40. Ввести с клавиатуры 10 целых чисел. Посчитать количество нечетных.
41. Вывести таблицу кубов первых 10-ти нечетных натуральных чисел.

    Пример оформления решения:

```
Число  Куб
-----------
  1      1
  3     27
  5    125
  7    343
  9    729
```

42. Вывести таблицу степеней двойки (от нулевой до десятой).

    Пример оформления решения:

```
Степень Результат
-----------------
  0        1
  1        2
  2        4
  3        8
  4       16
  5       32
```

43. Вывести на экран таблицу значений выражения $-2{,}4x^2 + 5x - 3$ в диапазоне от -2 до 2 с шагом 0,5.

    Пример:

```
    x    |    y
--------+-------
  -2.0  | -22.6
```

```
-1.5  | -15.9
-1.0  | -10.4
-0.5  |  -6.1
 0.0  |  -3.0
 0.5  |  -1.1
 1.0  |  -0.4
 1.5  |  -0.9
 2.0  |  -2.6
```

44. Ввести с клавиатуры последовательность из 5 вещественных чисел, и после ввода каждого числа выводить на экран среднее арифметическое полученной части последовательности.

For. Перебор со сравнениями

45. Введите с клавиатуры 10 натуральных чисел. Выведите на экран сумму четных чисел.
46. Введите с клавиатуры 10 натуральных чисел. Выведите на экран количество двузначных чисел.
47. Введите с клавиатуры 10 двузначных чисел. Выведите на экран сумму цифр тех из них, которые не делятся на 3.
48. Введите с клавиатуры 10 чисел. Выведите на экран количество тех из них, синус которых больше косинуса.
49. Введите с клавиатуры натуральное число N. Выведите на экран те натуральные числа, не превышающие N, которые делятся на 13 или на 7.

While-Repeat. Поиск

50. С клавиатуры вводятся числа, пока не будет введено отрицательное число. Посчитать их количество.
51. С клавиатуры вводятся числа, пока не будет введено число, делящееся на 5. Посчитать сумму тех из них, которые больше 10.
52. С клавиатуры вводятся натуральные числа, пока не будет введено не двузначное. Посчитать сумму цифр во всех двузначных числах.

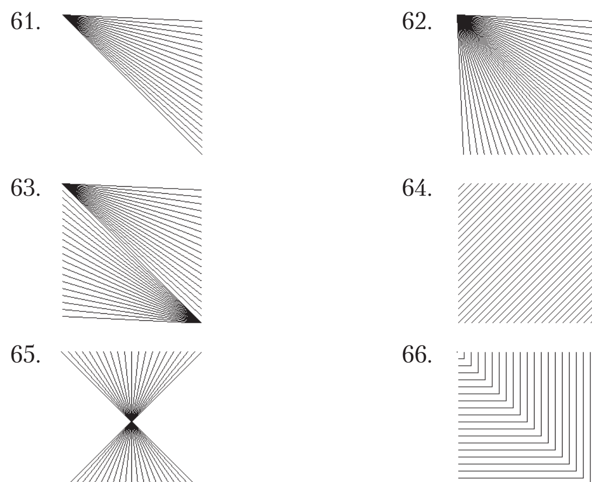
53. С клавиатуры вводится два натуральных числа — A и B. Посчитать частное и остаток от деления A на B, не используя операцию деления (нужно вычитать из первого второе и считать, сколько раз это удалось сделать).
54. С клавиатуры вводятся символы, пока не будет введена точка. Посчитать количество цифр (используйте в цикле вместо `readln` оператор `read`; после цикла поставьте `readln` без параметров).
55. С клавиатуры вводится натуральное число. Посчитать количество цифр в двоичном представлении числа (нужно делить число на 2 и считать остатки).
56. С клавиатуры вводится натуральное число. Посчитать двоичный вес числа (количество единиц в двоичном представлении числа).

While-Repeat. Ряды

57. Введите с клавиатуры целое число N ($N > 1$). Выведите на экран первые N чисел ряда: $\frac{1}{1}, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}, \cdots$ Посчитайте их сумму.
58. Введите с клавиатуры целое число N ($N > 1$). Выведите на экран первые N чисел ряда: $\frac{1}{1}, \frac{3}{2}, \frac{5}{4}, \frac{7}{8}, \frac{9}{16}, \frac{11}{32}, \cdots$ Посчитайте их сумму.
59. Введите с клавиатуры целое число N ($N > 1$) и вещественное число x. Выведите на экран первые N чисел ряда: $2, 4x, 6x^2, 8x^3, 10x^4, \ldots$ Посчитайте их сумму.
60. Введите с клавиатуры число ε ($0 < \varepsilon < 1$) и вещественное число x ($0 < x < 1{,}5$). Выведите на экран числа ряда: $\frac{1}{2}, \frac{x}{4}, \frac{x^3}{16}, \frac{x^5}{256}, \frac{x^7}{65\,536}, \ldots$, которые больше ε. Посчитайте их сумму.

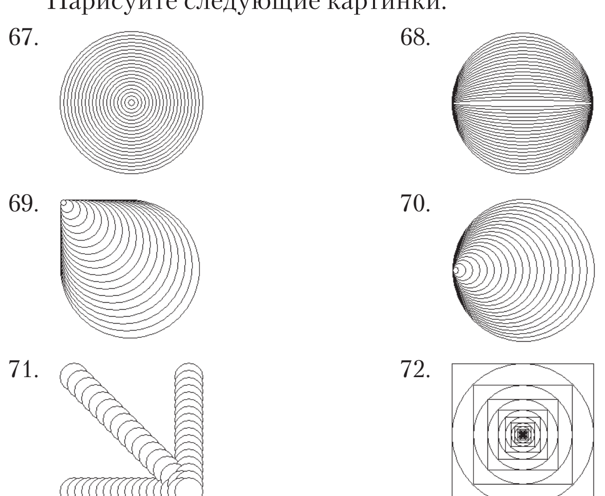
Графика. Прямые

Нарисуйте следующие картинки:



Графика. Окружности

Нарисуйте следующие картинки:



Массивы. Заполнение, вывод, сумма/количество

Для целочисленного массива (например, из 20 элементов):

73. Заполнить элементы массива последовательностью чисел: 2, 5, 8, 11, ...
74. Заполнить элементы массива последовательностью чисел: 2, 4, 8, 16, 32, ...
75. Заполнить элементы массива последовательностью чисел: 1, 3, 7, 15, 31, ...
76. Найти произведение всех элементов массива.
77. Найти среднее арифметическое всех элементов массива.
78. Найти среднее отличие элементов массива от их правого соседа. Например, для массива (5, 1, 3, 8) (из четырех элементов) имеется три пары соседних элементов (5 и 1, 1 и 3, 3 и 8). Отличия в парах составляют 4, 2, 5 соответственно. Среднее отличие равно $(4 + 2 + 5)/3 \approx 3{,}67$).
79. Обнулить все отрицательные элементы массива и посчитать количество остальных.
80. Все четные положительные элементы целочисленного массива уменьшить вдвое, все нечетные положительные увеличить на 2, а у всех остальных поменять знак.
81. Найти количество отрицательных элементов массива.
82. Найти среднее арифметическое всех нечетных элементов целочисленного массива.
83. Найти количество всех элементов целочисленного массива, которые не являются положительными двузначными числами.

Массивы. Перестановки

Для некоторого массива (например, из 20 элементов):

84. Развернуть массив в обратном порядке.
85. Поменять местами элементы массива в парах (из массива (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) нужно получить массив (2, 1, 4, 3, 6, 5, 8, 7, 10, 9)).

86. Сдвинуть все элементы массива на одну позицию влево (циклически). Первый элемент должен оказаться на месте последнего.
87. Сдвинуть все элементы массива на одну позицию вправо (циклически). Последний элемент должен оказаться на месте первого.
88. Развернуть обе половинки массива в обратном порядке. Считать, что массив имеет четное число элементов (из массива (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) нужно получить массив (5, 4, 3, 2, 1, 10, 9, 8, 7, 6)).
89. Поменять местами половинки массива (из массива (1, 2, 3, 4, 5, 6, 7, 8,9, 10) нужно получить массив (6, 7, 8, 9, 10, 1, 2, 3, 4, 5)).
90. Ввести два целых числа от 1 до 20 (первое меньше второго). Развернуть в обратную сторону часть массива, расположенную между элементами с такими номерами.
91. Сдвинуть все элементы массива на k позиций влево (циклически). Каждый элемент нужно переставлять не более двух раз. Разрешается использовать не более k дополнительных ячеек памяти.

Массивы. Поиск

Во всех заданиях этого блока считать неизвестным, имеется ли в массиве искомый элемент.

Для некоторого массива (например, из 20 элементов):

92. Найти номер первого положительного элемента массива.
93. Найти номер второго четного элемента массива.
94. Найти номер предпоследнего нечетного элемента массива.
95. Ввести с клавиатуры число. Найти номер элемента массива, равного этому числу.
96. Найти длину самой длинной последовательности одинаковых элементов.

Массивы. Проверки

Для некоторого массива (например, из 20 элементов):

97. Проверить, что в массиве все элементы положительны.
98. Проверить, что все элементы массива равны друг другу.
99. Проверить, есть ли в массиве элемент, равный заданному значению (ввести с клавиатуры).
100. Проверить, что массив упорядочен по возрастанию (каждый элемент массива больше своего левого соседа или равен ему).
101. Проверить, что в массиве все элементы кратны трем и больше нуля.
102. Проверить, что в массиве есть более чем два различных элемента.
103. Проверить, что в массиве нет ни одного одинакового элемента.

Массивы. Максимумы

Для некоторого массива (например, из 20 элементов):

104. Найти номер наибольшего элемента.
105. Найти номер элемента, наименее отличающегося от среднего арифметического всех его элементов.
106. Посчитать количество элементов, равных максимальному.
107. Найти номера элементов, равных максимальному.
108. Найти номер наибольшего отрицательного элемента.
109. Найти номер элемента массива, наиболее отличающегося от своих соседей (по сумме модулей разностей).
110. Найти номер элемента, значение которого меньше максимума, но больше всех остальных элементов.

Подпрограммы без параметров

111. Напишите процедуру, выводящую на экран вертикально 20 раз слово «Hello!»

112. Напишите функцию, вычисляющую количество двузначных чисел, у которых квадрат разницы между цифрами больше суммы цифр, и выводящую эти числа на экран.
113. Напишите функцию, вычисляющую количество трехзначных чисел, делящихся на 13 и не оканчивающихся на 3.
114. Напишите подпрограмму, которая выводит на экран таблицу умножения размером 9×9.
115. Напишите подпрограмму, которая запрашивает с клавиатуры последовательность целых чисел, пока не будет введен ноль, и возвращает среднее арифметическое положительных введенных чисел.
116. Напишите подпрограмму, которая сравнивает два введенных с клавиатуры числа и возвращает результат их сравнения (первое больше, меньше или равно второму).

Строки. Часть I

117. Ввести строку. Поменять местами ее левую и правую половины.
118. Ввести строку. Посчитать количество пробелов в ней.
119. Ввести строку. Удалить из нее все лишние пробелы (все подряд стоящие пробелы большие, чем один).
120. Ввести строку. Заменить все слова `Masha` на слова `Irisha`.
121. Ввести строку. Посчитать количество цифр в ней.
122. Ввести строку. Определить, есть ли в ней хоть одна заглавная латинская буква.
123. Ввести строку, состоящую из слов (последовательности символов строчной латыни и арабских цифр, разделенные пробелами). Первые и последние символы в словах, если это строчная латынь, преобразовать в заглавные.

Строки. Часть II

При выполнении заданий этого блока не разрешается использовать переменные числовых типов данных (в том числе `integer`, `real`) и функции преобразования строки в число.

124. Ввести строку, в которой содержатся две точки. Вывести на экран часть строки, находящуюся между этими точками.
125. Ввести строку. Вывести на экран эту же строку столько раз, сколько точек содержится в строке.
126. Ввести строку, состоящую из трехзначного числа, запятой и еще одного трехзначного числа (всего 7 символов). Вывести на экран результат сравнения этих двух чисел (больше, меньше или равно).
127. Ввести строку, состоящую из некоторого количества точек, знака «минус» и еще некоторого количества точек (меньшего первого). Вывести на экран «разницу» — столько точек, сколько останется, если из первого количества точек «отнять» второе количество.
128. Ввести строку, состоящую из двух чисел, разделенных запятой (каждое число произвольного размера). Вывести на экран, является ли первое число большим, чем второе.
129. Ввести строку, состоящую из трех чисел, разделенных запятыми. Вывести на экран большее из трех чисел.

Подпрограммы с параметрами. Часть I

130. Напишите процедуру, выводящую на экран указанное число раз указанную текстовую строку.
131. Напишите функцию `sign` (возвращает знак целого числа: +1, 0 или –1).
132. Напишите функцию, вычисляющую значение числа X, возведенного в степень Y.
133. Напишите процедуру, удваивающую значение числа.
134. Напишите процедуру, меняющую два числа местами.

135. Напишите функцию, возвращающую количество раз, которое одна строка встречается в другой строке.

Подпрограммы с параметрами. Часть II

136. Напишите процедуру, выводящую на экран все делители указанного числа.
137. Напишите функцию, проверяющую, является ли число простым.
138. Напишите функцию, вычисляющую количество трехзначных чисел, сумма цифр которых больше 10, лежащих в указанном диапазоне.
139. Напишите функцию, возвращающую количество цифр в десятичном представлении целого числа.
140. Напишите процедуру, «разворачивающую» целое число в обратном порядке (в десятичной записи).
141. Напишите функцию, проверяющую, является ли число палиндромом в десятичном представлении.

Подпрограммы с параметрами. Часть III

Напишите подпрограмму с параметрами, которая:

142. Из четырех чисел выдает число, наиболее близкое к среднему арифметическому.
143. Выдает номер старшего разряда двоичного представления натурального числа.
144. Вычисляет число ненулевых двоичных разрядов натурального числа.
145. Выполняет разворот двоичного представления натурального числа.
146. Вычисляет средний бит двоичного представления натурального числа (среди значащих цифр).
147. Меняет местами старшую и младшую половинки двоичного представления натурального числа.

148. Выполняет циклический сдвиг влево двоичного представления натурального числа (старший значащий бит (всегда =1) поставить в конец, остальные сдвинуть влево).

Файлы

Для некоторого текстового файла:

149. Посчитать число строк файла.
150. Найти самую длинную строку.
151. Посчитать число слов в файле. Слова считать разделенными одним или несколькими пробелами.
152. Породить новый файл с переставленным порядком строк (в парах).

    Пример:

    ```
    Было   Стало
      1      2
      2      1
      3      4
      4      3
      5      5
    ```

153. Посчитать количество строк файла, которые не содержат слово «begin».
154. Вывести на экран строки файла, в которых совпадают первый и последний символы.
155. Вывести на экран все строки файла развернутыми в обратную сторону.
156. Вывести на экран строки файла нечетной длины, в которых средний символ — цифра.
157. Вывести на экран все строки файла, удалив в них все цифры.
158. Вывести на экран строки файла, которые содержат не менее двух запятых.
159. Записать в другой файл только те строки исходного файла, в которых число цифр больше числа пробелов.
160. Посчитать, сколько раз в файле встречается отдельное слово «end» (слева и справа от слова находятся пробелы или начало/конец строки).

161. Для текстового файла (например, программы на Паскале) определить, соответствует ли каждому слову «begin» свое слово «end». Считать, что программа не содержит комментариев и апострофов. Не обязательно считать, что эти «begin» и «end» являются отдельными словами.

162. Вывести файл на экран со словами, записанными в обратном порядке. Слова считать разделенными одним или несколькими пробелами.

     Пример: Строку «Вася Петя Саша» вывести как «Саша Петя Вася».

Однонаправленный список

Задания этого блока выполняются последовательно, в одной программе. Каждое задание основывается на предыдущем.

163. Заполнить однонаправленный список случайным количеством (10–20) случайных целочисленных значений (на интервале –15...+15). Вывести получившийся список на экран.

164. Посчитать количество нечетных элементов списка. Вывести результат на экран.

165. После каждого нечетного элемента списка добавить элемент, который на один больше. Вывести получившийся список на экран.

166. Удалить все четные элементы списка. Вывести получившийся список на экран.

167. Перед каждым положительным элементом списка добавить элемент, равный нулю. Вывести получившийся список на экран.

168. Поменять местами соседние элементы списка (1–2, 3–4, ...). Вывести получившийся список на экран.

169. Найти самый большой и самый маленький элементы списка и увеличить на 50 все элементы, находящиеся между ними. Вывести получившийся список на экран.

170. Последовательно, строчка за строчкой, вывести на экран элементы списка, значения которых принадлежат первому, второму, третьему и т. д. десяткам натуральных чисел.

Рекурсия

171. Ввести натуральное число. Вывести на экран его двоичное представление.
172. Ввести натуральное число. Вывести старшую цифру его десятичного представления.
173. Ввести натуральное число. Вывести наименьшее число, которое является степенью двойки и которое не меньше данного числа.
174. Ввести натуральное число. Вывести число, в котором обратный порядок цифр.
175. Найти наибольший общий делитель двух натуральных чисел, введенных с клавиатуры.
176. Вывести на экран все возможные комбинации N-значного числа в k-й системе счисления. N и k — натуральные, вводятся с клавиатуры (пример для $N = 3$ и $k = 2$: 000, 001, 010, 011, 100, 101, 110, 111).
177. Вывести на экран все возможные комбинации N различных цифр. N — целое, от 2 до 9, вводится с клавиатуры (вывести все перестановки).
178. Ввести с клавиатуры натуральное число. Вывести все представления числа в виде суммы натуральных чисел, без повторений. Пример: $4 = 3 + 1 = 2 + 2 = 2 + 1 + 1 = 1 + 1 + 1 + 1$.

Денис Михайлович Ушаков, Татьяна Анатольевна Юркова

Паскаль для школьников

2-е издание

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Юрченко*
Ведущий редактор	*Ю. Сергиенко*
Литературный редактор	*О. Некруткина*
Художественный редактор	*Л. Адуевская*
Корректор	*Н. Першакова*
Верстка	*Е. Егорова*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2;
95 3005 — литература учебная.
Подписано в печать 10.04.13. Формат 60х90/16. Усл. п. л. 20,000. Тираж 3000. Заказ
Отпечатано в полном соответствии с качеством предоставленных издательством материалов
в ГППО «Псковская областная типография». 180004, Псков, ул. Ротная, 34.

А. Васильев

JAVA. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ. УЧЕБНОЕ ПОСОБИЕ. СТАНДАРТ ТРЕТЬЕГО ПОКОЛЕНИЯ

400 с., 16,5×23,5, обл.

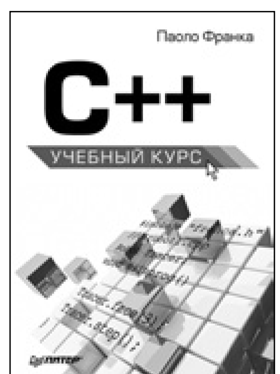
Учебное пособие предназначено для изучающих объектно-ориентированное программирование в вузе, а также для всех желающих самостоятельно изучить язык программирования Java.

В первой части книги излагаются основы синтаксиса языка Java. Материала первой части книги достаточно для написания простых программ. Во второй части описываются темы, которые будут интересны тем, кто хочет освоить язык на профессиональном уровне. Каждая глава книги содержит теоретический материал, иллюстрируемый простыми примерами, позволяющими подчеркнуть особенности языка программирования Java. В конце каждой главы первой части имеется раздел с примерами решения задач.

Учебное пособие соответствует Государственному образовательному стандарту 3-го поколения для специальностей «Информатика и вычислительная техника», «Информационные системы и технологии», «Прикладная информатика» и «Фундаментальная информатика и информационные технологии».

П. Франка

C++. УЧЕБНЫЙ КУРС. 2-Е ИЗДАНИЕ



496 с., 16,5×23,5, обл.

Язык C++ является в настоящее время одним из самых распространенных языков программирования, но одновременно и одним из самых трудных для изучения. Книга «C++: учебный курс» поможет быстро, эффективно и с наименьшими затратами освоить все основные приемы создания приложений на C++.

Для изучения всех возможностей языка требуются объемные руководства и справочники, но эта книга даст вам «стартовый толчок», поможет понять структуру языка, принципы объектно-ориентированного программирования, методику проектирования и создания приложений. Учебный материал, содержащийся в 26-ти уроках, основан на практических примерах и сопровождается исходным кодом программ. Для его освоения не требуется никакой предварительной подготовки. Книга может быть использована в качестве учебного пособия для студентов, изучающих язык C++.

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают эксклюзивный ассортимент компьютерной, медицинской, психологической, экономической и популярной литературы

РОССИЯ

Санкт-Петербург м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва м. «Электрозаводская», Семеновская наб., д. 2/1, корп. 1, 6-й этаж
тел./факс: (495) 234-38-15, 974-34-50; e-mail: sales@msk.piter.com

Воронеж Ленинский пр., д. 169; тел./факс: (4732) 39-61-70
e-mail: piterctr@comch.ru

Екатеринбург ул. Бебеля, д. 11а; тел./факс: (343) 378-98-41, 378-98-42
e-mail: office@ekat.piter.com

Нижний Новгород ул. Совхозная, д. 13; тел.: (8312) 41-27-31
e-mail: office@nnov.piter.com

Новосибирск ул. Станционная, д. 36; тел.: (383) 363-01-14
факс: (383) 350-19-79; e-mail: sib@nsk.piter.com

Ростов-на-Дону ул. Ульяновская, д. 26; тел.: (863) 269-91-22, 269-91-30
e-mail: piter-ug@rostov.piter.com

Самара ул. Молодогвардейская, д. 33а; офис 223; тел.: (846) 277-89-79
e-mail: pitvolga@samtel.ru

УКРАИНА

Харьков ул. Суздальские ряды, д. 12, офис 10; тел.: (1038057) 751-10-02
758-41-45; факс: (1038057) 712-27-05; e-mail: piter@kharkov.piter.com

Киев Московский пр., д. 6, корп. 1, офис 33; тел.: (1038044) 490-35-69
факс: (1038044) 490-35-68; e-mail: office@kiev.piter.com

БЕЛАРУСЬ

Минск ул. Притыцкого, д. 34, офис 2; тел./факс: (1037517) 201-48-79, 201-48-81
e-mail: gv@minsk.piter.com

Ищем зарубежных партнеров или посредников, имеющих выход на зарубежный рынок.
Телефон для связи: **(812) 703-73-73**. **E-mail:** fuganov@piter.com

---

Издательский дом «Питер» приглашает к сотрудничеству авторов. Обращайтесь по телефонам: **Санкт-Петербург** – **(812) 703-73-72**, **Москва** – **(495) 974-34-50**

Заказ книг для вузов и библиотек по тел.: (812) 703-73-73.
Специальное предложение – e-mail: kozin@piter.com

Заказ книг по почте: на сайте **www.piter.com**; по тел.: (812) 703-73-74
по ICQ 413763617

ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

У Вас есть свой сайт?
Вы ведете блог?
Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?
ЭТО ВПОЛНЕ РЕАЛЬНО!

СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!

*Зарегистрируйтесь на нашем сайте в качестве партнера по адресу **www.piter.com/ePartners***

Получите свой персональный уникальный номер партнера

*Выбирайте книги на сайте **www.piter.com,** размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт **www.piter.com**)*

ВНИМАНИЕ! *В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.*

С этого момента получайте 10% от стоимости каждой покупки, которую совершит клиент, придя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером. А если покупатель приобрел не только эту книгу, но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

Пример партнерской ссылки:
http://www.piter.com/book.phtml?978538800282 – обычная ссылка
http://www.piter.com/book.phtml?978538800282**&refer=0000** – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

Подробно о Партнерской программе ИД «Питер» читайте на сайте WWW.PITER.COM

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **postbook@piter.com**
- по телефону: **(812) 703-73-74**
- по почте: **197198, Санкт-Петербург, а/я 127, ООО «Питер Мейл»**
- по ICQ: **413763617**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.

С помощью банковской карты. Во время заказа Вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.

Электронными деньгами. Мы принимаем к оплате все виды электронных денег: от традиционных Яндекс.Деньги и Web-money до USD E-Gold, MoneyMail, INOCard, RBK Money (RuPay), USD Bets, Mobile Wallet и др.

В любом банке, распечатав квитанцию, которая формируется автоматически после совершения Вами заказа.

Все посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку Ваших покупок максимально быстро. Дату отправления Вашей покупки и предполагаемую дату доставки Вам сообщат по e-mail.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

ПИТЕР®

Нет времени

ходить по магазинам?

www.piter.com

Все книги издательства сразу
Новые книги — в момент выхода из типографии
Информацию о книге — отзывы, рецензии, отрывки
Старые книги — в библиотеке и на CD

И наконец, вы нигде не купите наши книги дешевле!