Министерство образования Республики Беларусь
Белорусский государственный университет
Механико-математический факультет
Кафедра веб-технологий и компьютерного моделирования

СОГЛАСОВАНО
Заведующий кафедрой
_____ В.М. Волков
«29» декабря 2022 г.

СОГЛАСОВАНО
Декан факультета
_____ С.М. Босяков
«26» января 2023 г.

СОГЛАСОВАНО
Председатель учебно-методической
комиссии факультета
_____ В.Г. Кротов
«26» января 2023 г.

Методы программирования

Электронный учебно-методический комплекс для специальности:
1-31 03 01 «Математика (по направлениям)»
направление специальности:
1-31 03 01-02 «Математика (научно-педагогическая деятельность)»

Регистрационный № 2.4.2-24/316

Авторы:
Расолько Г. А., кандидат физ.-мат. наук, доцент;
Кремень Е. В., кандидат физ.-мат. наук, доцент;
Кремень Ю. А., кандидат физ.-мат. Наук, доцент.

Рассмотрено и утверждено на заседании Научно-методического совета БГУ 15.02.2023 г., протокол № 5.

Минск 2023

УДК 004.42(075.8)
Р 242

Утверждено на заседании Научно-методического совета БГУ
Протокол № 5 от 15.02.2023 г.

Решение о депонировании вынес:
Совет Механико-математического факультета
Протокол № 5 от 26.01.2023 г.

Авторы:

Расолько Галина Алексеевна, кандидат физико-математических наук, доцент, доцент кафедры ВТиКМ ММФ БГУ;

Кремень Елена Васильевна, кандидат физико-математических наук, доцент, доцент кафедры ВТиКМ ММФ БГУ;

Кремень Юрий Алексеевич, кандидат физико-математических наук, доцент, доцент кафедры ВТиКМ ММФ БГУ.

Рецензенты:

кафедра алгебры, геометрии и математического моделирования Брестского государственного университета имени А.С. Пушкина (заведующий кафедрой Сендер А. Н., кандидат физико-математических наук, доцент);

Бровка Н. В., заведующий кафедрой теории функций ММФ БГУ, кандидат физико-математических наук, доктор педагогических наук, профессор.

Расолько, Г. А. Методы программирования : электронный учебно-методический комплекс для специальности: 1-31 03 01 «Математика (по направлениям)», направление специальности: 1-31 03 01-02 «Математика (научно-педагогическая деятельность)» / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень ; БГУ, Механико-математический фак., Каф. веб-технологий и компьютерного моделирования. – Минск : БГУ, 2023. – 226 с. : ил., табл. – Библиогр.: с. 225.

Электронный учебно-методический комплекс (ЭУМК) по учебной дисциплине «Методы программирования» предназначен для студентов специальности 1-31 03 01 «Математика (по направлениям)», направление специальности 1-31 03 01-02 «Математика (научно-педагогическая деятельность)». ЭУМК содержит тексты лекций, планы лабораторных занятий, перечень контрольных вопросов, тесты, списки рекомендованной литературы.

ОГЛАВЛЕНИЕ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА 6

1. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ 9

1 семестр 9

1.1. Введение 9
Содержание 9
Развитие языков компьютерного программирования 9
Эволюция Pascal 10
Free Pascal 10
1.2. Арифметика ЭВМ. Лабораторные занятия №№1-3. 11
Системы счисления 11
Переводы чисел из одной системы счисления в другую 12
Формы представления данных 14
1.3. Алгоритмизация 20
Алгоритм 20
Основные этапы решения задач на ЭВМ 21
1.4. Структурное программирование и точность программ 25
Основные конструкции структур управления 26
1.5. Средства алгоритмического языка Pascal 28
Базовые элементы языка Pascal 29
Общая структура Pascal–программы 32
1.6. Простые данные языка Pascal 33
Типы данных 33
Константы и переменные 34
Целочисленные данные 35
Вещественные данные 38
Выражения языка 40
Символьные данные 41
Логические данные 42
Данные адресного типа 42
Данные пользовательского типа 43
Данные перечислимого типа 43
Данные интервального типа 44
1.7. Простейшее определение процедур и функций. Файловая система 44
Простейшее определение процедур и функций 45
Параметры 47
О файловой системе 49
Файловый тип 49
Стандартные текстовые файлы 49
1.8. Базовые операторы Pascal 51
Ввод данных разных типов 51
Вывод данных разных типов 52

Простые операторы ... 54
Пустой оператор ... 54
Оператор безусловного перехода goto ... 54
Оператор вызова процедуры ... 54
Составной оператор ... 55
1.9. Структурные операторы ... 55
Операторы ветвления ... 55
Условный оператор ... 56
Оператор выбора ... 59
Операторы повторения ... 63
Итерационные алгоритмы высшей математики ... 70
1.10. Структуры данных и работа с ними ... 75
Порядковые типы ... 75
Множества ... 75
Массивы ... 80
Строки символов ... 85
Комбинированный тип «запись» ... 89
Записи с вариантами ... 93
Оператор присоединения with ... 96
Изменение (приведение) типов и значений ... 99
1.11. Процедуры и функции ... 102
Процедуры пользователя ... 103
Функции пользователя ... 103
Параметры ... 104
Принцип локализации ... 107
Побочный эффект ... 109
1.12. Рекурсия и итерация ... 110
Параметры без типа ... 113
Процедуры и функции как параметры ... 115
Переменные – процедуры и функции ... 117

2 семестр ... 119

1.13. Модули ... 119
Стандартные библиотечные модули ... 119
Модули пользователя ... 121
Подпрограммы в модулях ... 122
1.14. Файлы в языке Pascal ... 129
Файловые типы ... 129
Операции над файлами ... 129
Обработка ошибок ввода-вывода ... 135
Слияние двух отсортированных последовательностей ... 137
Текстовые файлы ... 142
Файлы без типа ... 145
1.15. Специальные средства Turbo Pascal. Модуль System ... 147
Указатели и динамические структуры данных ... 147

Программирование алгоритмов с использованием указателей .. 153
Разные варианты размещения матрицы в Heap 158
Проблема потерянных ссылок 167
Введение в связные динамические структуры данных 169
Алгоритмы работы с линейными списками 170
1.16. Модуль DOS. Модуль CRT 177
Модуль DOS 177
Модуль CRT 178
Видеодоступ 186
1.17. Модуль Graph 193
Графическое программирование 193
Ресурсы модуля GRAPH 194
Базовые процедуры и функции 197
Управление параметрами образов 200
Построение графических фигур 204
Работа с текстом 209
Экран и окно 213
Манипулирование фрагментами образов 214
Анимация 216
2. ПРАКТИЧЕСКИЙ РАЗДЕЛ 219
Программа лабораторных занятий 219
3. РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ 220
3.1. Перечень рекомендуемых средств диагностики 220
3.2. Примерный перечень заданий для управляемой самостоятельной работы студентов 220
3.3. Примерный перечень вопросов к зачету 221
3.4. Примерный перечень вопросов к экзамену 223
4. ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ 225
4.1. Список рекомендуемой литературы 225
4.2. Электронные ресурсы 225
4.3. Учебно-методическая карта учебной дисциплины 226

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Электронный учебно-методический комплекс (ЭУМК) по учебной дисциплине «Методы программирования» создан в соответствии с ОСВО 1-31 01 03-2018, типового учебного плана № G31-1-011/пр.тип. от 31.03.2021 г., учебных планов № G31-1-016/уч. от 25.05.2021 г., № G31-1-010/уч. ин. от 31.05.2021 г., № G31-1-015/уч.з. от 31.05.2021 г.

Учебно-методический комплекс преследует цель по оказании посильной помощи студентам в усвоении учебного и нормативного материала, сориентировать в подборе специальной литературы для подготовки к лабораторным и практическим занятиям по курсу «Методы программирования" в соответствиии с учебной программой учреждения высшего образования по учебной дисциплине для специальности первой ступени высшего образования 1-31 03 01 Математика (по направлениям), направление 1-31 03 01-02 Математика (научно-педагогическая деятельность). № УД-9969/уч 2021г. [Электронный ресурс]. – Режим доступа: https://elib.bsu.by/handle/123456789/270994 – Дата доступа: 24.02.2023.

Цель учебной дисциплины – формирование навыков решения различных типов задач на основе современных информационных технологий, а именно: развитие алгоритмического мышления, изучение современных методов программирования, приобретение навыков и освоение работы на современных вычислительных средствах (знакомство с современными методологиями приобретения знаний).

Задачи учебной дисциплины:

1. Развитие математического, логико-алгоритмического и программистского стилей мышления;
2. Формирование практических знаний и умений использования современных методов и систем программирования;
3. Овладение приемами и основами методологии структурного и модульного программирования;
4. Выработка творческого подхода к конструированию алгоритмов с целью развития аналитических и творческих способностей студентов.

В качестве базового учебного языка программирования выбран объектно-ориентированный язык Pascal, позволяющий осваивать классические приемы и современные технологии программирования.

Изучаются стандартные типы данных, управляющие структуры и операторы, вопросы процедурного и модульного программирования, работа с файлами. Основное внимание в курсе уделено не столько вопросу кодирования программы, сколько вопросу проектирования, где упор делается на современные технологии: проектирование сверху-вниз; модульное программирование, т.е. использование аппарата подпрограмм и модулей; проведение анализа эффективности участков программ и их оптимизация; широкое использование аппарата рекурсии. Всё вышеизложенное делается с целью привить некоторый стиль программирования. Полученные навыки далее

развиваются посредством обучения объектно-ориентированному программированию.

В результате освоения учебной дисциплины студент должен:

знать:

- методы решения научно-технических и информационных задач;
- современные информационные технологии;

уметь:

- решать типовые задачи математики и информатики;
- работать на современных вычислительных средствах;
- применять современные информационные технологии и методы реализации решения прикладных задач;

владеть:

- методами программирования задач в различных областях;
- современными технологиями разработки программ.

Дисциплина изучается в 1 и 2 семестрах дневной и заочной формы получения высшего образования по специальности 1-31 03 01 Математика (по направлениям), направление специальности 1-31 03 01-02 Математика (Научно-педагогическая деятельность). Всего на изучение учебной дисциплины «Методы программирования» отведено:

– для очной формы получения высшего образования – 210 часов, в том числе 140 аудиторных часов, из них: лекции – 70 часов, лабораторные занятия на персональных компьютерах – 60 часов, управляемая самостоятельная работа – 10 часов. Из них:

– в 1-м семестре: лекции – 36 часов, лабораторные занятия – 30 часов, управляемая самостоятельная работа – 6 часов.

Трудоемкость учебной дисциплины составляет 3 зачетные единицы.

Форма текущей аттестации – экзамен.

– во 2-м семестре: лекции – 34 часа, лабораторные занятия – 30 часов, управляемая самостоятельная работа – 4 часа.

Трудоемкость учебной дисциплины составляет 3 зачетные единицы.

Форма текущей аттестации – экзамен.

– для заочной формы получения высшего образования – 210 часов, в том числе 42 аудиторных часа, из них: лекции – 22 часа, лабораторные занятия – 20 часов.

Трудоемкость учебной дисциплины составляет 6 зачетных единиц.

Форма текущей аттестации – экзамен.

В структуру ЭУМК входит:

1. Теоретический раздел (включает краткий конспект лекций по учебной дисциплине).

2. Практический раздел.

3. Контроль самостоятельной работы студентов (содержит перечень контрольных мероприятий управляемой самостоятельной работы студентов; темы рефератов и вопросы для подготовки к зачету).

4. Вспомогательный раздел (включает список рекомендуемой литературы).

Работа студента с ЭУМК должна включать ознакомление с тематическим планом учебной дисциплины, представленным в учебной программе учреждения высшего образования, в которой можно получить информацию о тематике лекций, лабораторных занятий и рекомендуемой литературе. Для подготовки к лабораторным занятиям рекомендуется использовать материалы, представленные в теоретическом и практическом разделах ЭУМК, а также материалы для контроля самостоятельной работы студентов. В ходе подготовки к зачету целесообразно ознакомиться с требованиями к компетенциям по учебной дисциплине, изложенными в учебной программе учреждения высшего образования, а также перечнем вопросов к зачету.

1. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

В теоретическом разделе изложен краткий конспект лекций в соответствии с программой дисциплины.

1 СЕМЕСТР

1.1. Введение

Содержание

- Развитие языков компьютерного программирования.
- Эволюция Pascal.
- Free Pascal.

Развитие языков компьютерного программирования

Так как компьютер может обрабатывать информацию в системах счисления, отличных от десятичной (двоичная, троичная системы счисления), то для первых компьютеров программистам приходилось писать программы в кодах. Их называли программами на *машинном языке*.

Дальше, чтобы облегчить создание программ, был разработан язык *ассемблер*. В ассемблере машинные команды представлялись *мнемонически*-символьными инструкциями.

Когда программа на ассемблере написана, ее нужно преобразовать в программу на машинном языке.

Создали специальную программу – *транслятор ассемблера*, которая автоматически выполняла такой перевод.

Каждой команде ассемблера приблизительно соответствует одна команда машинного языка, поэтому ассемблер называют языком низкого уровня.

В зависимости от порядка выполнения программ, языки высокого уровня делятся на компилируемые и интерпретируемые.

Компилятор – это программа, которая автоматически преобразовывает (транслирует, компилирует) исходный код языка высокого уровня в машинный код и создает, таким образом, выполняемый файл (объектный код).

Интерпретатор – это программа, преобразующая (транслирующая, интерпретирующая) исходный код языка высокого уровня в машинный код шаг за шагом, это значит каждая команда (оператор) исходной программы преобразовывается интерпретатором и тут же выполняется компьютером, и так до тех пор, пока не закончится исходная программа на языке высокого уровня.

В дальнейшем тенденция облегчения для человека процесса создания программ осталась доминирующей.

В результате были разработаны *языки высокого уровня*, у которых программные конструкции подобны предложениям английского языка.

Примерами таких языков в наше время служат Fortran, Basic, Pascal, C, Java. Количество используемых языков программирования сейчас достаточно большое.

В последнее время разрабатываются объектно-ориентированные языки высокого уровня.

Примерами объектно-ориентированных языков являются C++, Java, Visual Basic, Object Pascal.

Эволюция Pascal

Язык программирования Pascal, созданный в 70-х годах швейцарским ученым и преподавателем Никлаусом Виртом в Цюрихе (Швейцария) как учебный язык компьютерного обучения программированию, назван в честь известного французского философа, математика, физика и писателя Блеза Паскаля.

Хорошая структурированность языка помогает привить начинающим программистам правильные навыки программирования.

Компанией Borland была разработана популярная версия языка – Turbo Pascal.

По мере развития версий операционных систем Windows и распространения концепции объектно-ориентированного программирования язык Pascal был расширен до Turbo Pascal for Windows и Object Pascal for Windows.

Следующим шагом было создание Delphi – среды разработки программ на Object Pascal.

В систему Delphi входят компилятор с Object Pascal, визуальная среда разработки, инструменты взаимодействия с базами данных и библиотека VCL (Visual Components Library – библиотека визуальных компонент).

Free Pascal

Free Pascal (полное название Free Pascal Compiler, часто используется сокращение FPC) – это свободно распространяемый компилятор языка Паскаль с открытым исходным кодом.

Важной особенностью данного компилятора является ориентация на совместимость с распространёнными коммерческими диалектами языка: Borland Pascal 7, Object Pascal и Delphi.

Free Pascal поддерживает компиляцию в нескольких режимах, обеспечивающих совместимость с различными диалектами и реализациями языка, например:

TP – режим совместимости с Turbo Pascal: совместимость практически полная, за исключением нескольких моментов, связанных с тем, что FPC компилирует программы для защищённого режима процессора, где невозможно прямое обращение к памяти, портам и т. д.

FPC – собственный диалект: соответствует предыдущему, расширенному дополнительными возможностями, такими как, например, перегрузка операций.

1.2. Арифметика ЭВМ. Лабораторные занятия №№1-3.

Содержание занятий

- Системы счисления.
- Переводы чисел из одной системы счисления в другую.
- Формы представления данных.
- Хранения чисел с фиксированной точкой.
- Хранения чисел с плавающей точкой.

Системы счисления

Под *системой счисления* будем понимать способ представления всякого числа при помощи некоторого алфавита символов, которые называются *цифрами*.

Система счисления (с/с) называется *позиционной*, когда одна и та же цифра имеет разное значение, соответствующее позиции цифры в последовательности цифр, образующих число.

Количество S разных цифр, используемых в позиционной системе счисления, называется ее *основанием*. Цифры, используемые в системе счисления для записи чисел, называются *базисными числами*. Эти цифры обозначают S целых чисел, обычно такого ряда: 0, 1, 2, 3, ..., (S–1). В системах счисления, где S>10, цифры большие, чем 9, обозначают буквами (таблица 1).

Таблица 1 – Базовые числа систем счисления.

с/с	Основание с/с	Базовые числа
10 с/с	S = 10	0, 1, 2, 3, …, 9
2 с/с	S = 2	0, 1
16 с/с	S = 16	0, 1, …, 9, A, B, C, D, E, F

В общем случае в позиционной системе с основанием S любое число x может быть представлено в виде выражения, зависящего от основания S:

$$x = \varepsilon_r S^r + \varepsilon_{r-1} S^{r-1} + ... + \varepsilon_1 S^1 + \varepsilon_0 S^0 + \varepsilon_{-1} S^{-1} + ... + \varepsilon_{-t} S^{-t} + ... \qquad (1)$$

где в качества коэффициентов ε_i могут быть любые из S цифр, используемых в данной системе счисления.

Вместо вида (1) принято писать короче:

$$x = \varepsilon_r \varepsilon_{r-1} ... \varepsilon_1 \varepsilon_0 . \varepsilon_{-1} ... \varepsilon_{-t} ... \qquad (2)$$

Позиции цифр в числе, которые отсчитываются от разделителя целой и дробной части, называются *разрядами*. Разделителем целой части и дробной в математике служит запятая, а в программировании – точка.

В позиционной системе счисления значение каждого разряда больше чем значение соседнего справа разряда в число раз, равное основанию S системы.

Переводы чисел из одной системы счисления в другую

Общий метод перевода небольших чисел из одной системы счисления в другую такой: *представить число в виде* (1) *и выполнить действия в новой системе счисления.*

Представление чисел разных систем счисления дается в следующей таблице (таблица 2).

Таблица 2 – Представление чисел разных систем счисления.

10 с/с	**16 с/с**	**2 с/с**	**Для перевода 2 с/с ↔ 16 с/с**	**Для перевода 2 с/с ↔ 8 с/с**
0	0	0	0000	000
1	1	1	0001	001
2	2	10	0010	010
3	3	11	0011	011
4	4	100	0100	100
5	5	101	0101	101
6	6	110	0110	110
7	7	111	0111	111
8	8	1000	1000	
9	9	1001	1001	
10	A	1010	1010	
11	B	1011	1011	
12	C	1100	1100	
13	D	1101	1101	
14	E	1110	1110	
15	F	1111	1111	
16	10	10000		
...	...	...		

Из этой таблицы видно, что любая 16-я цифра требует четыре двоичные разряда, а 8-я – три разряда. Тогда получим следующие правила перевода.

Правило 1. *Чтобы перевести число из 16 с/с в 2 с/с, нужно любую цифру расписать в 2 с/с по четыре разряда (с 8 с/с в 2 с/с – по три разряда; с 4 с/с в 2 с/с – по два разряда), а потом отбросить незначащие нули.*

Правило 2. *Чтобы перевести число из 2 с/с в систему счисления с основанием* 2^k, $k = 3,4$, *нужно от запятой (точки) налево и направо выделить группы по* k *цифр (дополняя при необходимости незначащими нулями крайние группы) и каждой группе поставить в соответствие определенную цифру новой системы.*

Перевод чисел из систем счисления $p = 2^k \to q = 2^n$ можно осуществить через 2-ю с/с.

Перевод целых положительных чисел из системы счисления с основанием “p” в систему счисления с основанием “q”

Пусть x_p – целое положительное число в системе счисления с основанием “p”. В соответствии с формулой (1) в системе счисления с основанием “q”, число x_p может быть представлено в виде:

$$x_p = a_n \cdot q^n + a_{n-1} \cdot q^{n-1} + ... + a_0 \equiv (...(a_n \cdot q + a_{n-1}) \cdot q + ... + a_1) \cdot q + a_0. \quad (3)$$

Отсюда для нахождения коэффициентов при степенях q в формуле (3) получаем следующее правило перевода:

Правило 3. Чтобы перевести целое число из системы счисления с основанием “p” в систему счисления с основанием “q” нужно выполнять следующие действия. В системе счисления с исходным основанием “p” разделить x на q. Первый остаток – младшая цифра (a_0). Затем полученное частное разделить на q. Остаток есть следующая цифра (a_1). Новое частное делим на q. Этот процесс продолжаем до тех пор, пока не получим частное, меньшее чем q. Эта частное есть старшая цифра (a_n). Все действия выполняются в системе с основанием p. Полученные остатки записываем в новой системе счисления с основанием “q”.

Для отрицательных чисел переводятся их модули, затем учитывается знак числа.

Перевод правильных дробей из системы счисления с основанием “p” в систему счисления с основанием “q”

Пусть y_p – правильная дробь в системе счисления с основанием “p”. Аналогично предыдущему имеем

$$y_p = a_{-1} \cdot q^{-1} + a_{-2} \cdot q^{-2} + ... + a_{-m} \cdot q^{-m} \equiv q^{-1}\left(a_{-1} + q^{-1}\left(a_{-2} + q^{-1}...\right)\right)$$

Правило 4. Чтобы перевести правильную дробь из системы счисления с основанием “p” в систему счисления с основанием “q”, нужно: исходное число умножить на q (в с/с с исходным основанием “p”).

Целая часть полученного числа есть цифра a_{-1}.

Полученную дробную часть умножить на q. Целая часть полученного числа есть цифра a_{-2} и т. д.

Все действия выполняются в системе с основанием p.

В конце перевода записать полученные цифры $(a_{-1}, a_{-2}, ...)_p$ соответствующими цифрами в новой системе счисления с основанием “q”.

Процесс перевода дробной части необходимо остановить при достижении одного из следующих условий:

1) когда получили произведение, равное нулю (число перевелось точно);
2) когда получили произведение, которое уже встречалось раньше, значить нашли период;
3) когда получили нужное количество цифр в числе, которое переводится.

Перевод смешанных дробей

Для перевода целой части числа применяется правило 3, для перевода дробной части числа – правило 4.

Формы представления данных

Числа можно представлять в форме с фиксированной точкой (2) и в форме с плавающей точкой. Представление числа с плавающей точкой в общем случае имеет вид

$$x = q \cdot s^p \qquad (4)$$

где q – мантисса числа, s^p – характеристика числа x (s – основание характеристики, p – порядок).

Мантисса – дробь с фиксированной точкой со знаком, *порядок* – целое число со знаком.

Порядок p определяет положение точки в числе x.

Чтобы получить число в форме с плавающей точкой, нужно перевести число в нужную систему счисления, а потом записать его по правилу (4):

$$\left(a_{n-1}...a_1a_0.a_{-1}a_{-1}...a_{-m}\right)_p = 0.a_{n-1}...a_{-m} \times \underbrace{p^n}_{=10_p^n} = 0.a_{n-1}...a_{-m} \times 10_p^n$$

Арифметические действия над числами с плавающей точкой требуют кроме выполнения операции над мантиссой определенных операций над порядками (сравнения, сложения, вычитания).

Формы представления чисел в персональном компьютере

Информация в памяти ЭВМ хранится и обрабатывается в двоичной системе счисления.

Неделимой наименьшей единицей хранения информации является один *бит*, т. е. двоичный разряд, который может иметь значение 0 или 1.

Группа из 8 бит называется *байтом*. Вся оперативная память состоит из байтов, которые нумеруются с нуля.

Последовательность в 1024 байт называется 1 килобайтом (1 Кбайт = 2^{10} байтов); 1 мегабайт = 1024 Кбайтов (1 Мбайт = 2^{20} байтов), 1 гигабайт =1024 Мбайтов (1 Гбайт = 2^{30} байтов), 1 терабайт =1024 Гбайтов (1 Тбайт = 2^{40} байтов), 1 петабайт =1024 Тбайт (1 Пбайт = 2^{50} байтов), 1 эксабайт =1024 Пбайт (1 Эбайт = 2^{60} байтов) и др.

Адресом любого данного считается адрес (номер) самого первого байта поля памяти, выделенной для его хранения.

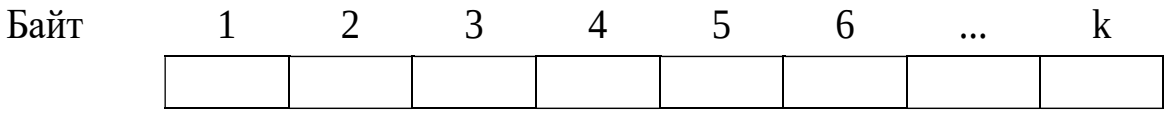
Форма записи данных в памяти ЭВМ называется *внутренним представлением данного*.

В ЭВМ применяют две формы представления чисел: с *фиксированной* и с *плавающей точкой*.

Рассмотрим формы представления чисел в IBM совместимых компьютерах.

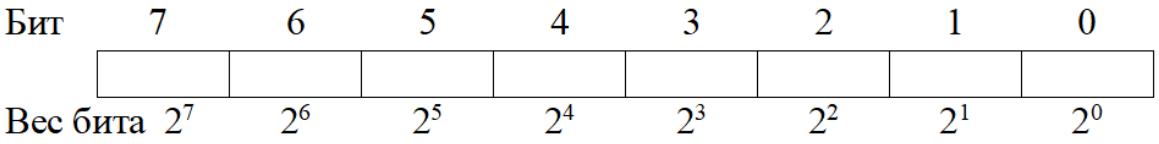
Хранение чисел с фиксированной точкой

Для хранения информации в оперативной памяти служит *ячейка*. Существуют ЭВМ, у которых ячейка имеет постоянную длину, в ПК – переменную, наименьшая длина – 1 байт. Различают ячейки на 2, 4, 6, 8, 10 байт. Ячейка в 2 байта называется *словом*, в 4 байта – *двойным словам*, в 1 байт – *полусловом*. Будем считать, что байты размещаются так (k=2, 4, 6, 8, 10):



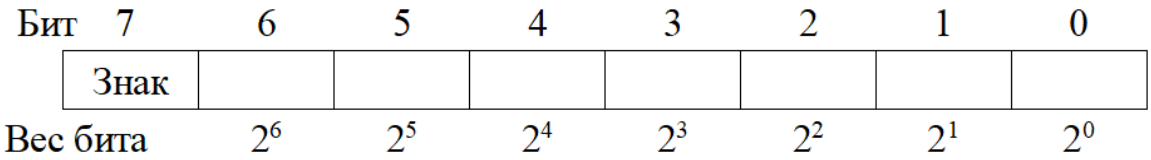
При представлении чисел с фиксированной точкой местоположение точки условно фиксируется в определенном месте ячейки относительно разрядов числа.

Каждый бит имеет свой вес в байте. Когда в байте точка зафиксирована справа и используется беззнаковая форма хранения целых положительных чисел, то биты имеют вес в соответствии со следующей схемой:

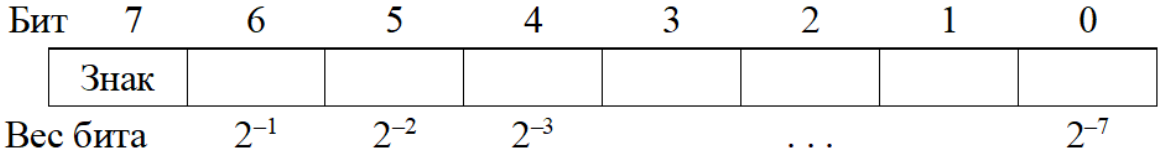


Этим методом представляют целые числа.

Используют два варианты представления целых чисел: со знаком и без знака. В первом случае для кода знака выделяют “знаковый” разряд (обычно крайний слева) и хранят там 0 или 1 по схеме: $0 \rightarrow +$, $1 \rightarrow -$



Если местонахождение точки фиксируется перед старшим разрядом, тогда вес разрядов в байте будет другим:



В этом формате могут быть представлены числа – правильные дроби.

Хранение целых чисел

Для представления целых чисел в форме с фиксированной точкой со знаком в ЭВМ применяют *прямой*, *обратный* и *дополнительный* коды.

Общая идея построения кодов такая. Код трактуется как число без знака, а диапазон представляемых кодами чисел без знака разбивается на два поддиапазона. Один из них дает положительные числа, а второй – отрицательные. Разбиение выполняется таким образом, чтобы принадлежность к диапазону определялась максимально просто. Удобно формировать коды так, чтобы значение старшего разряда указывало на знак числа.

Прямой код (*пр*) двоичного числа G, представляемого в $n-$ разрядной сетке ($n=8,16,32$), определяется как

$$G_{пр}=\begin{cases}G, \text{ при } G\geq 0\\ A+|G|, \text{ при } G<0\end{cases},$$

где A – величина, равная весу старшего разряда сетки (для целых чисел $A=2^{n-1}$).

Фактически в прямом коде, хранится модуль числа, а в старшем разряде стоит знаковый бит: 0 – для положительных чисел, 1 – для отрицательных. Диапазон чисел в прямом коде: $0\leq G<A$ – для положительных, старший разряд равен 0, $0<|G|<A$ – для отрицательных, старший разряд равен 1.

$$\left\{\begin{array}{l}0\leq G_{пр}<A \quad \text{– для положительных чисел,}\\ A\leq G_{пр}<2A \text{ – для отрицательных чисел.}\end{array}\right\}$$

Обратный код двоичного числа G, представляемого в $n-$ разрядной сетке ($n=8,16,32$), определяется как

$$G_{обр}=\begin{cases}G & \text{при } G\geq 0,\\ B-|G| & \text{при } G\leq 0,\end{cases}$$

где B – величина наибольшего числа без знака, который размещается в n-разрядной сетке ($n=8,16,32$). $B=2^n-1$ – для целых чисел.

Диапазон чисел в обратном коде: $0\leq|G|<B=2^{n-1}$.

По этому определению обратный код отрицательного числа представляет собой дополнение модуля исходного числа к наибольшему числу без знака, который помещается в разрядную сетку. В связи с этим получение обратного кода отрицательного числа сводится к получению инверсии (замены 0 на 1, а 1 на 0) $n-$ разрядного кода модуля этого числа. Значит, знаковый бит имеет 0 для положительного числа и 1 – для отрицательного. Недостаткам такого представления является то, что число 0 и (–0) имеют разное представление.

Дополнительный код:

$$G_{доп}=\begin{cases}G & \text{при } G\geq 0,\\ C-|G| & \text{при } G<0,\end{cases},$$

где C – величина, равная весу разряда, который идет за старшим разрядом используемой разрядной сетки. Для целых чисел $C=2^n=B+1$.

Диапазон чисел в дополнительном коде: $0 \le G < A$ – для положительных, старший разряд равен 0; $0 < |G| \le A$ – для отрицательных, старший разряд равен 1. Для целых отрицательных чисел $G_{доп} = G_{обр} + 1$.

В дополнительном коде хорошо выполняются операции сложения и вычитания целых чисел.

Замечание. Когда происходит переполнение, то на это реагирует знаковый разряд.

Алгоритм представления отрицательного числа в дополнительном коде

Число переводится в 2 с/с; дополняется нулями до n разрядов; выполняется инверсия двоичных цифр; к коду добавляется число 1.

Замечание. В компьютере целые положительные числа хранятся в прямом, а целые отрицательные – в дополнительном коде (таблица 3).

Таблица 3 – Тип данных и диапазон чисел.

Тип данных	**Размер в байтах**	**Диапазон чисел**	**Способ хранения чисел**
Byte	1	0..255	Хранятся только неотрицательные числа в прямом коде.
Word	2	0..65535	
Shortint	1	–128..127	Неотрицательные числа хранятся в прямом коде, отрицательные – в дополнительном.
Integer	2	–32768..32767	
Longint	4	–2147483648.. 2147483647	

Хранения чисел с плавающей точкой

Так как арифметические действия над числами с плавающей точкой требуют кроме выполнения операции над мантиссой определенных операций над порядками (сравнения, сложения, вычитания), то для упрощения операций над порядками их сводят к действиям над целыми положительными числами без знака. К порядку p добавляют целое число – *смещение*. Обычно, смещение $A = 2^{k-1}$, где k – число двоичных разрядов, используемых для записи модуля порядка. Тогда $p_{см} = p + A > 0$ и $p_{см}$ называют *характеристикой числа*.

При фиксированном числе разрядов мантиссы любое число представляется в ЭВМ с наиболее возможной точностью нормализированным числом.

Используют две формы нормализации.

Число $x = q \cdot s^p$ называется *нормализированным*, когда мантисса –

1) правильная дробь: $\frac{1}{s} \le |q| < 1$,

2) $1 \le |q| < s$.

Например,

1. $(1101.0101)_2 = \begin{cases} 0.11010101 \cdot 2^4_{10} \to 0.11010101 \cdot 10^{100}_2 \\ 1.1010101 \cdot 2^3 \to 1.1010101 \cdot 10^{11}_2 \end{cases}$ *или* .
2. $(0.000101)_2 = 0.101 \cdot 2^{-3} = 1.01 \cdot 2^{-4}$.

В IBM совместимых персональных компьютерах, как правило, используется форма нормализации числа: $1 \le |q| < 2$. При такой нормализации старшую цифру мантиссы можно не хранить.

В памяти нужно хранить два числа: мантиссу и порядок, а основание системы счисления $p = 2$ – известно. Под число отводится ячейка – четное количество байт. Ячейку условно разбивают на две части. В одной размещают "сдвинутый" порядок числа, а во второй – мантиссу (обычно цифры после точки как условно целое). Под знак мантиссы также отводится 1 бит.

Использование "предполагаемого" (спрятанного) старшего разряда мантиссы приводит к необходимости представления чисел с нулевой мантиссой особым кодам, ведь нулевая мантисса не отличается от мантиссы, равной 1/2. Таким кодам взяли код, равный в компьютере всем нулям разрядной сети.

От конкретной ЭВМ зависит количество байт, отводимых под число; пропорция, в которой разбивается ячейка для хранения мантиссы и порядка; последовательность размещения мантиссы и порядка.

Чем больше бит отведут под порядок, тем больший получится диапазон представляемых в компьютере чисел; чем больше бит отведут под мантиссу, тем точнее будут представлены числа в компьютере.

Замечание 1. Данные, хранящиеся в ЭВМ в форме с плавающей точкой, почти всегда представлены с погрешностью и только приблизительно равны исходному числу. Погрешность обусловлена ограничением на длину мантиссы (на количество разрядов для мантиссы).

Замечание 2. Под порядок также отводится ограниченное количество разрядов, значит диапазон чисел с плавающей точкой ограничен: $min \le |x| \le Max$. Значения *min* и *Max* также зависят от конкретной ЭВМ.

Форматы чисел с плавающей точкой

Размеры памяти, выделяемой под числа с плавающей точкой, существенно зависят от аппаратной реализации компьютера.

Для IBM PC/AT числа с плавающей точкой занимают поле (ячейку) в 4 байта (`Single`), 6 байт (`Real`), 8 байт (`Double`), 10 байт (`Extended`).

Характеристика – это смещенный код порядка с отрицательным нулем. По стандарту смещение порядка равно $\left(2^{k-1} - 1\right)$, где k – количество разрядов, отведенных для хранения кода порядка. Значение порядка лежит в интервале $\left[-\left(2^{k-1} - 1\right), -0\right] \cup \left[1, \left(2^{k-1} - 1\right)\right]$. Для типа `Real` смещение равно $2^{k-1} + 1$.

Код $\underbrace{11...1}_{k}$ – не используется, т.к. зарезервирован для указания переполнения порядка или на потерю точности мантиссы.

При этом получаются следующие диапазоны представляемых чисел:

- при 4-байтовом хранении k = 8, $p_{см} \in [-127,127]$, что соответствует диапазону числа $\pm 10^{-38} \div \pm 10^{38}$, представляемого с погрешностью 2^{-23};

- при 6-байтовом хранении k = 8, $p_{см} \in [-127,127]$, что соответствует диапазону числа $\pm 10^{-38} \div \pm 10^{38}$, представляемого с погрешностью 2^{-39};

- при 8-байтовом хранении k = 11, $p_{см} \in [-1023,1023]$, что соответствует диапазону числа $\pm 10^{-308} \div \pm 10^{308}$, представляемого с погрешностью 2^{-52};

- при 10-байтовом хранении k = 15, $p_{см} \in [-16383,16383]$, что соответствует диапазону числа $\pm 10^{-4932} \div \pm 10^{4932}$, представляемого с погрешностью 2^{-64}.

Схема хранения чисел в памяти следующая:

Знак мантиссы

Single	±	Характеристика (*l*)			Нормализованная мантисса (*f*)		
номера битов	31	30	…	23	22	…	0

$\Rightarrow$ число расшифровывается так: $v = (-1)^{\pm} 2^{l-127} (1.f)$, если $0 < l \le 255$.

Знак мантиссы

Real	±	Нормализованная мантисса (*f*)			Характеристика (*l*)		
номера битов	47	46	…	8	7	…	0

$\Rightarrow$ число расшифровывается так: $v = (-1)^{\pm}\, 2^{l-129}\,(1.f)$, если $0 < l \le 255$.

Знак мантиссы

Double	±	Характеристика (*l*)			Нормализованная мантисса (*f*)		
номера битов	63	62	…	52	51	…	0

$\Rightarrow$ число расшифровывается так: $v = (-1)^{\pm} \cdot 2^{(l-1023)} \cdot (1.f)$, если $0 < l < 2047$.

Знак мантиссы

Extended	±	Характеристика (*l*)			Нормализованная мантисса (*f*)		
номера битов	79	78	…	64	63	…	0

$\Rightarrow$ число расшифровывается так: $v = (-1)^{\pm} \cdot 2^{(l-16383)} \cdot (0.f)$, если $0 < l < 32765$.

Во всех формах представления чисел $v = 0$, при $l = 0$.

1.3. Алгоритмизация

Содержание

- Алгоритм.
- Основные этапы решения задач на ЭВМ.
- Тестирование программ.

Алгоритм

Слово «алгоритм» по сути является синонимом слов «способ, рецепт» и т. д. Возникла оно в средние века, когда европейцы познакомились со способами выполнения над числами, записанными в арабской системе счисления, арифметических действий (сложения и умножения), описанными в книге Китаб аль-джебр валь-мукабала («Книга о сложении и вычитании») автора Абу Абдуллаха Мухаммеда ибн Муса аль-Хорезми. Таким образом, слово «алгоритм» оказывается европеизированным произношением слов «аль Хорезм».

Программа является некоторым *алгоритмом*, записанным на языке, который переводится на язык ПК.

Язык программирования – это совокупность средств и правил представления алгоритма в виде, приемлемом для компьютера. Основу языков программирования составляют *алгоритмические языки*.

Если возвратиться к понятию «алгоритм», то нужно отметить, что существует много определений этого понятия. Мы остановимся на следующем определении.

Алгоритм – предписание исполнителю выполнить точно определенную последовательность действий, направленных на достижение заданной цели или решение поставленной задачи.

Существуют разнообразные методики и технологии разработки алгоритмов.

Разработка компьютерной программы – длинный и трудоемкий процесс. Чтобы окончательный вариант программы работал правильно и содержал как можно меньше ошибок, программисты явно или нет, придерживаются полного цикла разработки программы.

Основные этапы решения задач на ЭВМ

Как бы далеко по своей сложности не стояли друг от друга задачи, например, решение квадратного уравнения или управления космическим кораблем, их решение на ЭВМ имеет ряд общих этапов. Обычно это такие этапы:

1. Постановка задачи.
2. Анализ, формализованное описание задачи, выбор математической модели. Разработка методов решения и определение ограничений на поставленную задачу.
3. Выбор или разработка алгоритма и запись его формальными средствами.
4. Программирование решения задачи на одном из языков программирования.
5. Тестирование и отладка программы.
6. Решение задачи на ЭВМ.

При решении конкретных задач некоторые этапы могут отсутствовать, а другие быть даже сложно разрешимыми.

Этап 1. Постановка задачи выполняется заказчиком, и она на первых порах может быть и не совсем конкретно алгоритмическая.

Этап 2. Анализ задачи включает определение входных и выходных данных, выявление возможных ограничений на их значения и обычно завершается формализованным описанием задачи, которое зачастую предполагает ее математическую формулировку.

Этап 3. Выбор или разработка алгоритма и метода решения задачи имеют огромное значение для успешной работы над программой. Точно продуманный алгоритм решения задачи - необходимое условие эффективного программирования.

Для формализации алгоритма существуют разные средства:

- запись на родном языке;

•запись на формализованном языке – запись на псевдокоде;

•графические средства, используемые программистом (структурные схемы, структурограммы);

•графические средства, которые использует компьютер при создании *p*-схем.

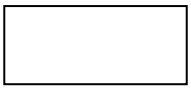
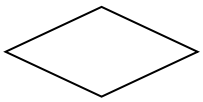
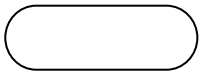
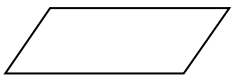
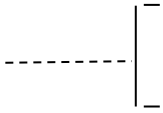
Когда программирование происходит на языках высокого уровня, структурная схема используется как язык сверх высокого уровня. Очень подробная схема даже вредна, а вот принципиальная, которая задает последовательность действий, может и сложных, – очень полезна.

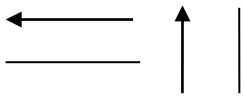
Мы будем в дальнейшем более применять структурные схемы и структурограммы.

Рассмотрим некоторые условные графические символы схем алгоритмов и программ, которые будем использовать в структурных схемах.

В таблицы приведены некоторые блоки и функции, наделенные им (таблица 4).

Таблица 4 – Блоки и их функции.

Обозначение блока	**Функция**
	Выполнение операций, в результате которых изменяется значение данных
	Выбор направления выполнения алгоритма в зависимости от некоторых условий
	Выполнение операций, которые изменяют команды или группы команд
	Начало или конец схемы программы
	Использование ранее созданных и отдельно описанных алгоритмов
	Указание связи между прерванными линиями потока, связывающими символами
	Обмен данными между внешней и оперативной памятью
	Ввод-вывод данных, носителем которых служит бумага
	Связь между элементами схемы и толкованием

Обозначение блока	Функция
	Естественное - сверху вниз или слева направо (без стрелок) и очевидное со стрелками

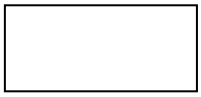
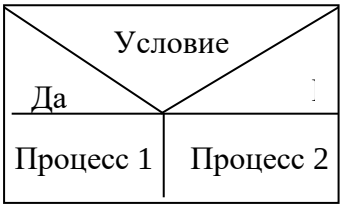
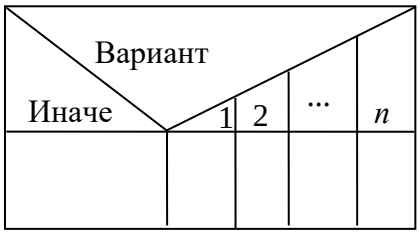
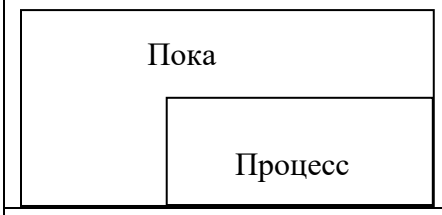
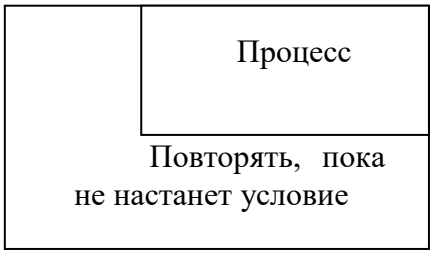
В блоках позволяется писать пояснительные действия. Например, $\boxed{x := 1}$ это такой процесс: переменной *x* присвоить значение 1, (знак «:=» – присвоить значение, вместо «=» – равно).

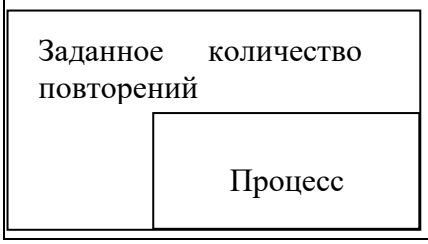
Недостаток блок-схем в том, что они занимают много места и поэтому появилось еще другие средства, например, *диаграммы Насси-Шнейдермана* или *структурограммы*.

Структурограммы – это вспомогательные средства для графического представления алгоритмов. Они дают хорошую систему описания и понимания программ и *используются для иллюстрации процесса передачи управления в программе*. Поэтому основные конструкции структурограммы – это процессы, а более простой символ – прямоугольник. Вся структурограмма – это прямоугольник, который разделяется на процессы – прямоугольники. Структурограмма позволяет компактно на одном листе сконструировать общую последовательность действий, которая приводит к решению задачи. Управление идет сверху-вниз.

Основные конструкции структурограмм (таблица 5).

Таблица 5 – Основные конструкции структурограмм.

Обозначение символа	Функция	Операторы языка
	Любая группа действий, образующая блок	**begin** … **end**
	Выбор направления выполнения алгоритма в зависимости от некоторого условия	**if** … **then** … **else**
	Выбирается вариант дальнейшего действия	**case** … **of** … **end**
	Цикл с предусловием	**while** … **do**
	Цикл с пост условием	**repeat** … **until**

	Цикл на заданное количество повторений	**for** … **to** … **do**
		
		for … **downto** … **do**

Блок-схемы требуют меньшей квалификации при их разработке. При использовании же структурограмм уже нужно знать о линейных и не линейных процессах, циклах с пред-, пост- условиями, или цикле на известное количество повторений.

В каждой из форм представления алгоритма есть свои преимущества и недостатки. Структурограммы держат разработчика в более жестких рамках. Писать по ним программы проще, потому что современные алгоритмические языки это, как правило, языки структурного программирования и в них существуют специальные операторы, которые приспособлены к

соответствующим конструкциям структурограммы, это: **begin** … **end; if** … **then** … **else;** case … **of** … **end; while** … **do; repeat** … **until; for** … **to**(**downto**) … **do** и другие.

Этап 4. Программа на алгоритмическом языке состоит из инструкций – операторов. Каждый язык имеет свои свойства и ориентацию на определенные классы задач. Если имеется возможность выбора языка, нужно осмыслить, какой язык больше всего подходит для решения поставленной задачи.

Этап 5. Тестирование и отладка программ представляют собой очень важные составляющие процесса разработки программы. *Тестирование* – это процесс выполнения алгоритма с целью определения в нем наличия ошибок, *отладка* – процесс локализации и исправления ошибок.

Этап 6. Когда ошибки все исправлены, начинается этап эксплуатации программы. При этом могут пригодиться описания, как постановки задачи, так и алгоритма, и текста программы.

1.4. Структурное программирование и точность программ

Содержание

- Концепции разработки алгоритмов.
- Основные конструкции структур управления.
- Алгоритмизация.
- Примеры на прием «пошаговой детализации».

Концепции разработки алгоритмов

Первое, что требуется от алгоритма решения задачи, это правильно реализовать задачу.

Второе – нужна такая реализация, которая является легкой для понимания, простой для доказательства правильности алгоритма и удобной для модификации.

Популярной методикой разработки алгоритма решения задачи является *структурное программирование сверху вниз*, когда задача осмысливается в целом, потом она разбивается на подзадачи, которые известно уже как алгоритмизировать средствами структурного программирования.

Современные языки программирования имеют операторы, которые хорошо отображаются диаграммами Насси-Шнейдермана, или структурограммами. Блок-схемы, подчиненные таким же условиям, также называются *структурными блок-схемами*. Структурные блок-схемы разрабатываются на базе шести структур управления.

Структурная блок-схема – это блок-схема, которая может быть выражена как композиция из шести элементарных схем, или структур управления.

Доказательство правильности алгоритма – это один из самых трудных и утомительных этапов *программирования*. Поэтому «прозрачность» отдельных действий разрешит легче проследить за правильностью всех действий по решению поставленной задачи.

Пользуясь управляющими конструкциями уже при разработке алгоритма легко следить за правильностью его частей, а этим и доказывать правильность всего алгоритма.

Это неизбежно приводит к обязательному требованию – разрабатывать (проектировать) программу так же, как проектируется любое устройство. Наиболее общая тактика разработки программ состоит в *декомпозиции* всей задачи на отдельные более простые подзадачи, которые приводят к решению всей задачи. Затем *процесс декомпозиции* распространяется на подзадачи и т. д. до тех пор, пока не получатся подзадачи, легко реализуемые на языке программирования.

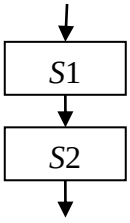
Декомпозиция задачи – это итерационный процесс, который не исключает обращение назад.

Важной особенностью всех структур управления является то, что каждая из них имеет один вход и один выход. Значит и вся блок-схема будет иметь это свойство, поэтому мы с начала алгоритма выйдем на конец его.

Для каждых из предложенных конструкций в структурном алгоритмическом языке существуют определенные операторы: **`if … then … else, if … then, case … of … end,`** `while …` **`do, repeat … until, for …`** **`to`**`(downto) …` **`do`**.

Основные конструкции структур управления

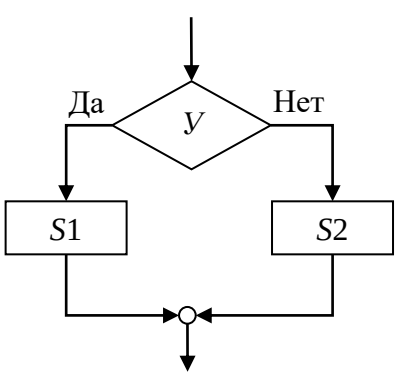
Следование:



*S*1 – процесс 1,

*S*2 – процесс 2.

Альтернатива:

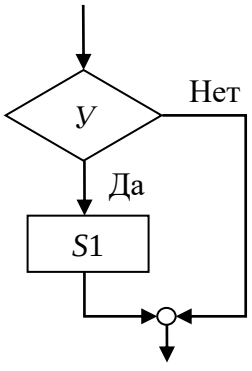


У – условие,

*S*1 – процесс 1,

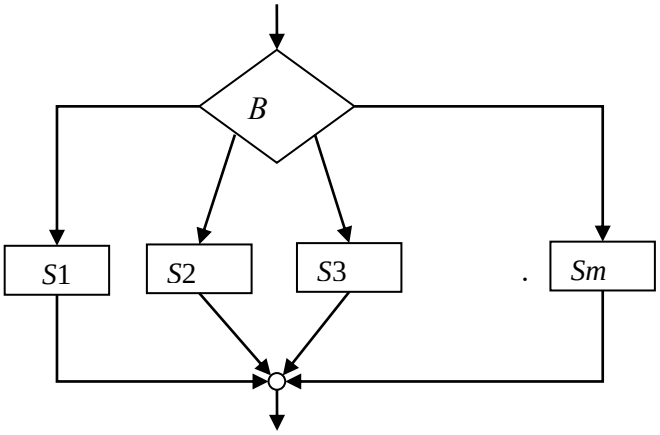
*S*2 – процесс 2.

(оператор **`if … then … else`**)



(оператор **`if … then`**)

Выбор:



B – вариант выбора,

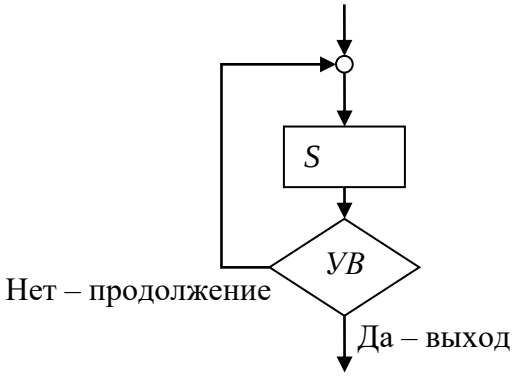
*S*1 – процесс 1,

. . .

Sm – процесс *m*.

(оператор **`case`** … **`of`** … **`end`**)

Повторение (***цикл***) ***с постусловием***:

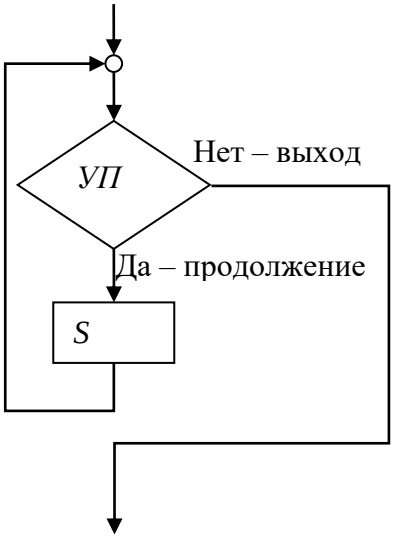


S – процесс,

УВ – условие выхода из цикла.

(оператор **`repeat … until`**)

Повторение (***цикл***) ***с предусловием***:

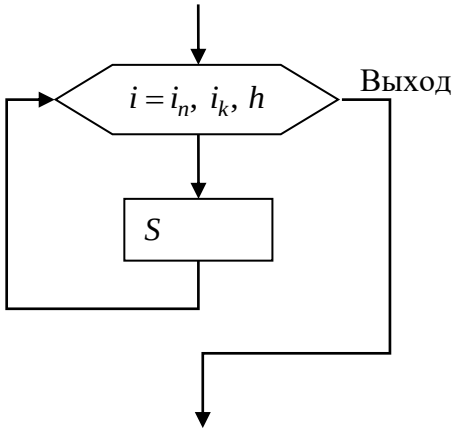


УП – условие продолжения цикла,

S – процесс.

(оператор **`while … do`**)

Цикл с перечислением (***параметром***):



i – параметр цикла,

i_n – начальное значение параметра цикла,

i_k – конечное значение параметра цикла,

h – шаг изменения параметра цикла,

S – процесс.

(оператор **`for … do (downto)`**)

Алгоритмизация

На стадии разработки алгоритмов и программ существенную помощь оказывают методы проектирования:

- *проектирование сверху вниз (нисходящее программирование);*
- *модульное программирование*;
- *проектирование снизу вверх* (*восходящее программирование*);
- *структурное кодирование*.

1.5. Средства алгоритмического языка Pascal

Содержание

- Общая характеристика алгоритмических языков.
- Базовые элементы языка Pascal:
  - алфавит,
  - лексическая структура языка,
  - общая структура Pascal–программы.

Общая характеристика алгоритмических языков

Основная цель, которая ставится при разработке алгоритмического языка, заключается в том, чтобы предложить некоторое формализованное средство общения между человеком и машиной, предназначенное для выполнения алгоритмов.

Каждый язык охватывает какую-то *ограниченную область приложения*, так как определенные ограничения накладывают требования компактности языка, удобства записи алгоритмов, понятности при его изучении, определенная простота создания транслятора, а также парк ЭВМ, которые будут его использовать.

Каждый алгоритмический язык характеризуется своей ориентацией на определенные типы данных и на указанные действия над данными. Элементы данных могут группироваться в структуры. Действия над данными выполняются с помощью *операторов* – предложений языка.

Для обозначения конкретных объектов алгоритмического языка используется понятие *переменной* величины.

Фундаментальным действием в любом алгоритмическом языке является действие *присваивания значения*, которое изменяет значение некоторой переменной. Присваивать значение можно при помощи *оператора* «:=» или вводом, а можно и косвенно.

Базовые элементы языка Pascal

Алфавит

Текст Pascal-программы представляет собой последовательность строк, состоящих из символов, образующих алфавит языка.

Строки программы заканчиваются специальными управляющими символами, которые не входят в алфавит. Максимальная длина строки – 126 символов.

Алфавит состоит из следующих символов*:*

- больших и малых букв и символа «подчеркивание», который отнесен к буквам: A, ..., Z, a, ..., z, _. Буквы используются для формирования идентификаторов и служебных слов (обратите внимание, что здесь отсутствуют буквы кириллицы);
- десяти арабских цифр от 0 до 9: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Цифры используются для записи чисел и идентификаторов;
- двадцати двух специальных символов и пробела: + – * / => < . , ; : @ '() [] {} # $ ^ (нет символа !, % и некоторых других).

Специальные символы используются для конструирования знаков операций, выражений, комментариев, а также как синтаксические разделители.

Лексическая структура языка

Символы из алфавита языка используются для построения базовых элементов Pascal-программ – лексем.

Лексема (лексический элемент) – минимальная единица языка, которая имеет самостоятельный смысл.

В языке Pascal существуют следующие классы лексем: 1) служебные слова; 2) идентификаторы; 3) изображения констант; 4) знаки операций; 5) разделители; 6) директивы компилятора.

1. *Служебные* (*зарезервированные*) слова.

Это ограниченная группа слов, построенных из букв.

Каждое служебное слово представляет собой неделимое образование, смысл которого зафиксирован.

Служебные слова нельзя использовать для других целей.

Приведем примеры некоторых служебных слов и охарактеризуем их (таблица 6):

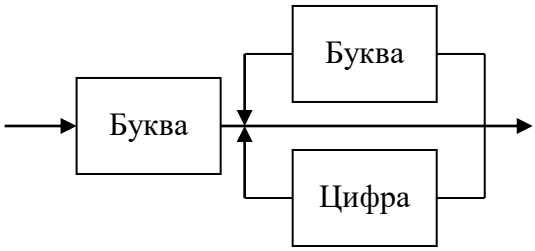
Таблица 6 – Служебные слова.

Название	Перевод	Толкование
absolute	абсолютно	Опция (режим)
and, or, xor	и, или, исключающее или	Операции
array	массив	Массив
assembler	ассемблер	Опция подпрограммы
begin, end	начало, конец	Операторные скобки
case, of	выбор из	Вариант
const	константы	Секция констант
constructor, destructor	создать, разрушить	Типы процедур
div, mod	-	Операции целочисленного деления
for, do, to, downto, repeat, while, until	для, выполнять, до, до, повторять, пока, до тех пор пока	Для создания операторов циклов
if, then, else	если, тогда, иначе	Для создания оператора выбора
external	внешний	Опция
file	файл	-
forward, function, procedure, program	впереди, функция, процедура, программа	Служебные слова
goto	перейти к	Оператор
implementation, interface	реализация, совокупность средств интерфейс	При описании внешнего модуля
inline, interrupt	–	Опции подпрограмм
in	в	Операция
label	метка	Секция меток
nil	–	Константа
not	Отрицание не	Операция
object	объект	Тип
private	приватный	Секция
record	запись	Составной элемент с компонентами разных типов
set of	множество из	Оператор
shl, shr	–	Операции
string	строка	Тип данного

Название	Перевод	Толкование
type	тип	Секция
unit, uses	модуль, используем	Служебные слова
var	переменные	Секция
with	вместе с	Оператор

2. *Идентификаторы* (*имена*). Вводятся для обозначения в программе переменных, констант, типов, меток, процедур, функций и формируются из букв и цифр в соответствии со следующей диаграммой.

Идентификатор:



В языке Pascal соответствующие прописные и строчные буквы в идентификаторах и служебных словах *не различаются*.

Идентификаторы вводятся в программу при помощи описаний.

В языке существуют так называемые стандартные идентификаторы для обозначения заранее определенных разработчиками языка типов данных, констант, процедур, функций. Их лучше использовать по их назначению без каких-либо изменений (например, `Integer`, `Pi`, `Maxint`, `Sin`, `Cos` и др.).

3. *Изображения констант*. Эта группа лексем обозначает неизменные объекты: числа, строка символов и др. (`13.14`; `'OX–OX'`, `$F7F0`, `#55`).

Знаки операций. Они формируются из одного или нескольких специальных символов и предназначены для задания действий по преобразованию данных и подсчета значений выражений. Примеры операций: <>, <=,> =, *, /, –, @, ^, **div**, **mod**, **and**, **or**, **xor**, **not**, **shl**, **shr**.

4. *Разделители*. Они также формируются из специальных символов и в основном используются, чтобы повысить наглядность текстов программ. К разделителям относятся ; : () = и др. *Пробел* используют для отделения лексем тогда, когда их слитное написание изменит содержание программы. Количество пробелов не ограничено.

5. *Комментарий* – это последовательность символов необязательно из алфавита языка (т.е. могут быть и русские буквы), которые заключены в фигурные скобки {...} или разделители вида (* ... *). например:

{комментарий (* вложенный комментарий *)
комментарий на двух строках}

Комментарии могут находиться между любыми двумя лексемами программы.

6. *Директивы компилятора.* Они задают *или* отменяют заданный режим компиляции или дают другую информацию компилятору. Синтаксис: {$имя_директивы...} . Эти конструкции так же, как и комментарии, заключаются в фигурные скобки, но пишутся по специальному синтаксису.

Например:

{$R+}, {$R–} – разрешение или запрет проверки границ скалярных данных или индексов массивов;

{$Q+}, {$Q–} – разрешение или запрет проверки переполнения целочисленных данных.

Общая структура Pascal-программы

Программа на языке Pascal всегда состоит из двух основных частей:

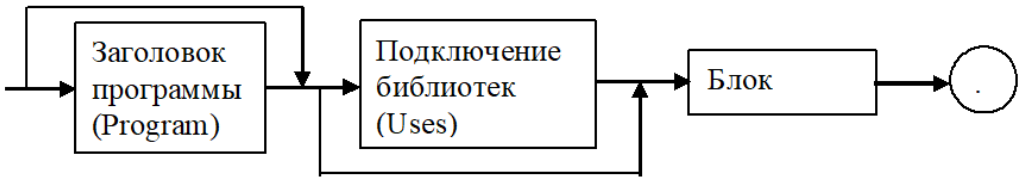
- *описания данных* (сведений), с которыми оперируют в программе, и
- *описания последовательности действий*, которые необходимо выполнить по обработке данных.

Действия представляются операторами языка – *исполняемые операторы*, данные описываются посредством *операторов описаний* (описаний).

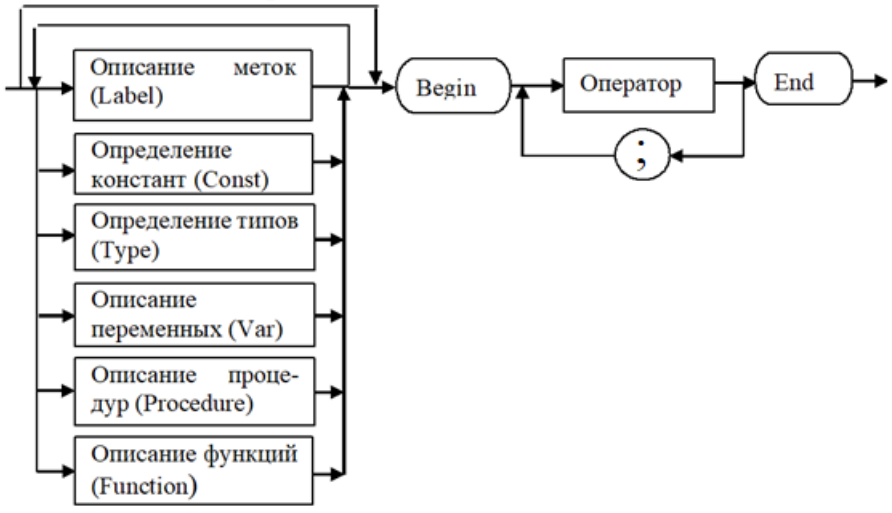
Описания данных текстуально предшествуют описанию действий и должны содержать упоминания всех объектов, использующихся в действиях (операторах).

Объекты – это метки, константы, типы, переменные, процедуры и функции.

Синтаксическая диаграмма программы на языке Pascal



Синтаксическая диаграмма блока



1.6. Простые данные языка Pascal

Содержание

- Типы данных.
- Константы и переменные:
  - абсолютные переменные,
  - типизированные константы,
- Целочисленные данные:
  - битовая арифметика,
  - действия битовой арифметики.

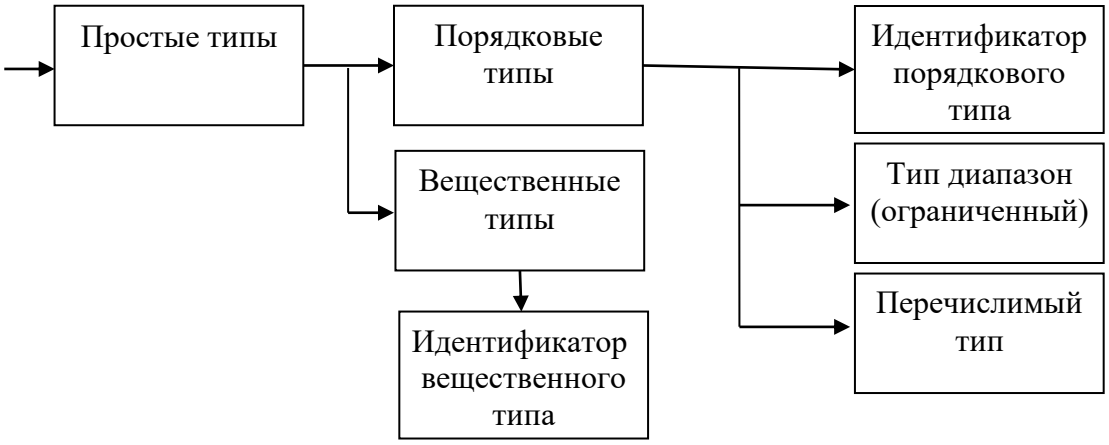
Типы данных

Тип данных определяет множество значений, которые могут принимать данные программы, и множество операций, допустимых над этими данными.

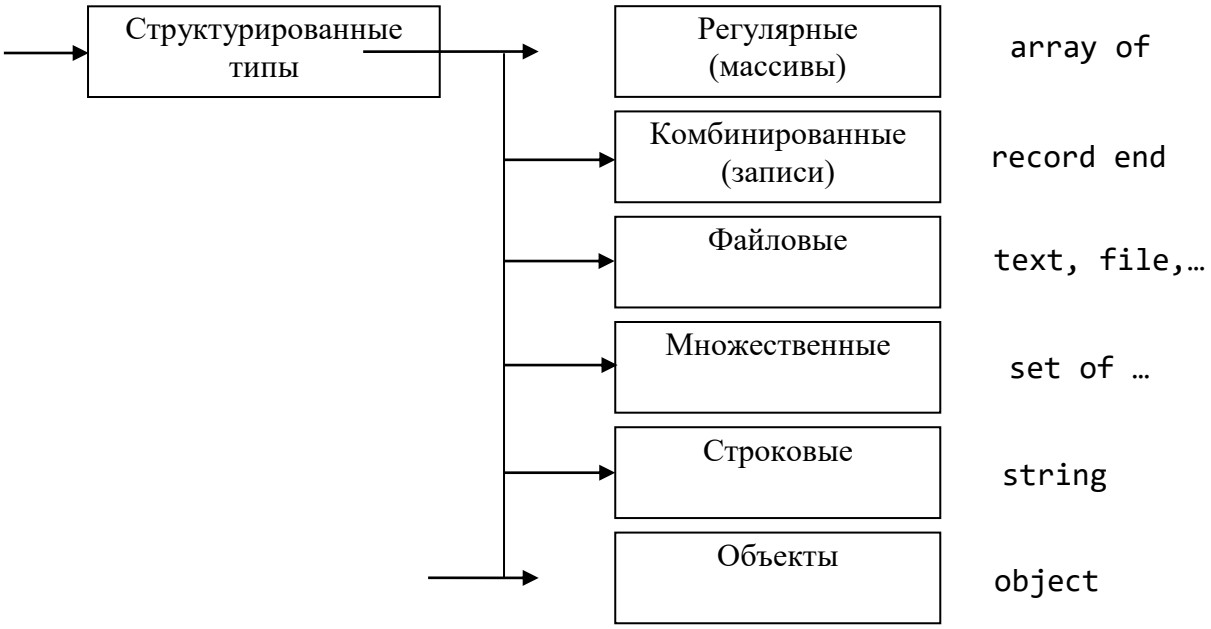
Базовыми в системе типов являются *простые* (*скалярные*) *типы. Стандартные* скалярные типы делятся на четыре группы*:* целочисленные типы, вещественные типы, символьный тип, логический тип .

Система типов языка Pascal следующая:

Первая группа:



Вторая группа:



Третья группа:

– образуются из любых других типов (как ^базовый тип)

Четвертая группа:

– позволяют работать с подпрограммами как с переменными

Константы и переменные

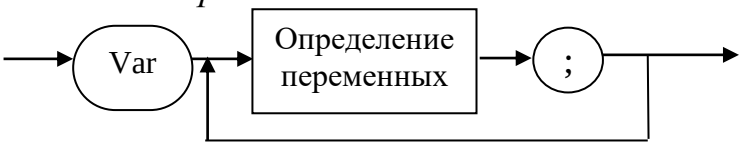
Любая программа имеет смысл, если она обрабатывает какие-либо данные. Как и другие языки программирования, Pascal интерпретирует данные как *константы* и *переменные*. Константы могут быть именованными, тогда их значения устанавливаются в секции **`const`** описательной части программы.

Именованные константы и переменные определяются идентификаторами (именами). К ним можно обращаться по именам, чтобы, например, получить текущее значение.

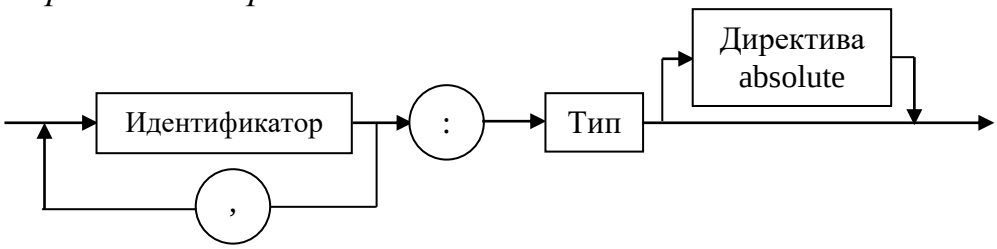
В Pascal существует ряд констант, которые можно использовать без предварительного определения. Это, например, `Pi`, `true`, `false`, `nil`, `Maxint` (они описаны в служебных программах).

Переменные получают свои значения в процессе выполнения программы. Каждая переменная и константа принадлежат к определенному (оговоренному) типу данных. Тип константы автоматически распознается компилятором по ее написанию без предварительного описания. Тип переменной должен быть описан. Переменные описываются в секции `var`.

Описание переменных:



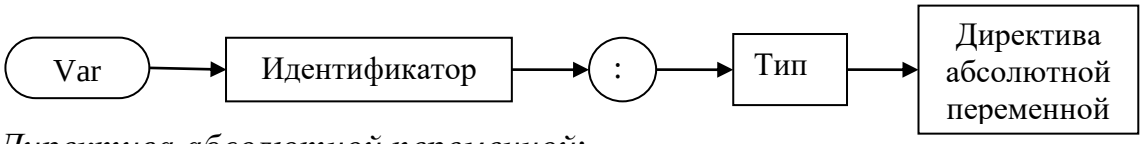
Определение переменных:



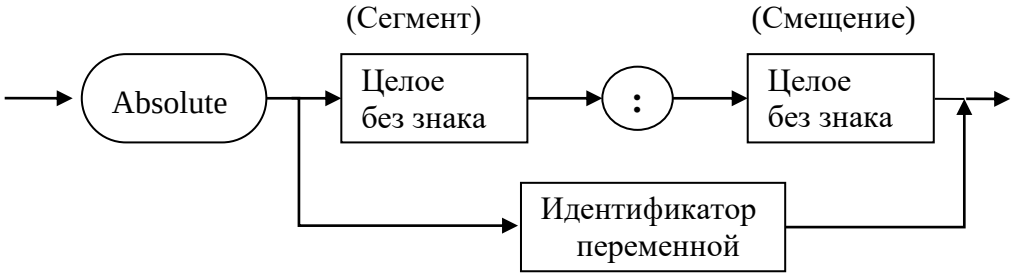
Абсолютные переменные

Переменные можно описать так, что они будут располагаться по определенному адресу в памяти, и тогда они называются *абсолютными*.

Объявление абсолютной переменной



Директива абсолютной переменной:



Первая форма с директивой `ABSOLUTE` содержит адрес переменной, заданный как сегмент и смещение. Обе константы не должны выходить за пределы диапазона от `$0000` до `$FFFF` (от 0 до 65535).

Типизированные константы

Формат определения типизированной константы:

```
Const        Идентификатор : тип = значение;
```

Пример определения типизированной константы:

```
Const        Max : Word = 10000;
```

Целочисленные данные

Эта группа типов обозначает множество целых чисел в различных диапазонах. Существует пять целых типов, которые отличаются допустимым диапазоном значений и размером занимаемой оперативной памяти. Их характеристики и названия приведены в таблице (таблица 7):

Таблица 7 – Целые типы и их диапазоны.

Целый тип	**Диапазон**	**Размер памяти**	**Особенность**
`Shortint`	-128 .. 127	1 байт	Со знаком
`Integer`	-32768 .. 32767	2 байта	
`Longint`	-2147483648 .. 2147483647	4 байта	
`Byte`	0 .. 255	1 байт	Без знака
`Word`	0 .. 65535	2 байта	

Над целыми данными определены следующие операции, дающие целый результат: *, +, –, но / – деление – дает вещественный результат.

`div` – деление нацело (с усечением), откидывается дробная часть. Например, выполнение операции `17 div 3` дает результат `5, 17 div 0` приводит к ошибке.

`mod` – выделение остатка от деления двух целых операндов. Например: `17 mod 2` дает результат `1`.

Кроме арифметических операций, к данным целого типа применяют операции отношений: <, >, =, <>, <=, >=. Результатом операций отношений является **`true`** (истина) или **`false`** (ложь).

Значения целых типов могут отображаться в программе двумя способами: в 10-й с/с и 16-й с/с. В последнем нетрадиционном случае цифры, старшие 9, обозначаются латинскими буквами от A до F (можно малыми), а в начале ставится символ $ (знак доллара).

Целый результат дают также следующие стандартные функции (аргумент от имени функции отделяется круглыми скобками):

$abs(x)$, $sqr(x)$, $trunc(x)$, $round(x)$.

Логический (булевский) результат дает функция $odd(x)$, x – целого типа. Если x нечетное, то результат **`true`**, иначе – **`false`**.

В следующих функциях аргумент может быть целочисленными, но результат будет вещественный:

$\sin(x)$, $\cos(x)$, $\arctan(x)$, $\ln(x)$, $\exp(x)$, $sqrt(x)$.

Процедуры `Dec(x[,n])` и `Inc(x[,i])` переводят целочисленный аргумент в целочисленный.

Битовая арифметика

Битовая, или поразрядная, арифметика хорошо развита в языке Pascal. Необходимость в ней возникает, когда нужно работать не с десятичными значениями чисел, а с их двоичным представлением. Битовые операции позволяют сравнивать отдельные биты двух чисел, выделять отдельные фрагменты в числе, заменять их.

Битовые операции к вещественным числам не применяются. Они применяются только к данным целого типа – `Byte, Shortint, Word, Integer, Longint`.

Общая формула для значений данных беззнаковых типов `Byte` (1 байт) и `Word` (2 байта) имеет следующий вид:

`Byte`: значение $= \beta_7 \cdot 2^7 + \beta_6 \cdot 2^6 + \beta_5 \cdot 2^5 + ... + \beta_1 \cdot 2^1 + \beta_0$;

`Word`: значение $= \beta_{15} \cdot 2^{15} + \beta_{14} \cdot 2^{14} + \beta_{13} \cdot 2^{13} + ... + \beta_1 \cdot 2^1 + \beta_0$.

Значения разрядов β_0, β_1, ... равные или 0, или 1, соответственно умножаются на вес разряда. Отсюда получаем диапазон представления:

`Byte`: $\underset{7\,6\,5\,4\,3\,2\,1\,0}{\boxed{11111111}} \Rightarrow 2^8 - 1 = 255$ (8 бит)

`Word`: $\boxed{11111111|11111111} \Rightarrow 2^{16} - 1 = 65535$ (16 бит)

Внутреннее отличие имеют представления целых типов со знаками: `Shortint, Integer, Longint`. Самый левый бит отводится под знак числа, для отрицательного числа он равен 1, для положительного – 0.

Можно запомнить формулу перевода из 2-й с/с в другую систему счисления для типов `Shortint` (1 байт), `Integer` (2 байта), `Longint` (4 байта):

`Shortint`: значение $= -\beta_7 \cdot 2^7 + \beta_6 \cdot 2^6 + \beta_5 \cdot 2^5 + ... + \beta_1 \cdot 2^1 + \beta_0$;

`Integer`: значение $= -\beta_{15} \cdot 2^{15} + \beta_{14} \cdot 2^{14} + \beta_{13} \cdot 2^{13} + ... + \beta_1 \cdot 2^1 + \beta_0$;

`Longint`: значение $= -\beta_{31} \cdot 2^{31} + \beta_{30} \cdot 2^{30} + \beta_{29} \cdot 2^{29} + ... + \beta_1 \cdot 2^1 + \beta_0$.

Отсюда диапазон представления чисел:

а) положительных Integer: $\boxed{01111111|11111111} \Rightarrow 2^{15} - 1 = 32767$;

б) отрицательных Integer (наименьшее получим, если ничего не будем добавлять): $\boxed{10000000|00000000} \Rightarrow -2^{15} = -32768$;

в) положительных Shortint: $\boxed{01111111} \Rightarrow 2^7 - 1 = 127$;

г) отрицательных Shortint: $\boxed{10000000} \Rightarrow -2^7 = -128$.

Действия битовой арифметики

Первая группа поразрядной арифметики – это логические операции над битами (таблица 8):

Таблица 8 – Логические операции над битами.

Операции	**Название**	**Форма записи**	**Приоритет**	**Тип**
not	Поразрядное отрицание	**not** A	1 (высший)	Унарная
and	Логическое умножение (и)	A1 **and** A2	2	Бинарная
or	Логическое сложение (или)	A1 **or** A2	3	
xor	Исключающее “или”	A1 **xor** A2	4 (низший)	

not – поразрядное отрицание –значение каждого бита изменяет на противоположное.

and, or, xor – логические операции, выполняемые поразрядно в соответствии со следующей таблицей истинности (таблица 9):

Таблица 9 – Таблица истинности.

A	B	A and B	A or B	A xor B	A xor B xor B
0	0	0	0	0	0
0	1	0	1	1	0
1	0	0	1	1	1
1	1	1	1	0	1

Примечание. Операция **and** в основном нужна для двух целей:

1) проверить наличие конкретных битов или

2) сбросить некоторые из них в нуль (занулить).

Логическое сложение **or** с успехом используется при установке значения в 1 (включение) отдельных битов двоичного представления целых чисел: `A := A or (16+2+1)` – включили четвертый, первый и нулевой биты, ведь 2^4=16, 2^1=2, 2^0=1.

Исключающее “или” **xor** – сравнение по модулю 2 – возвращает 0, если оба операнда равны, и 1 – если нет.

Следующая группа поразрядных операций – *циклические сдвиги.*

shl	Циклический сдвиг на *N* позиций влево (*l*)	`A shl N`
shr	Циклический сдвиг на *N* позиций вправо (*r*)	`A shr N`

Приоритет операций – как в **and**. В данных типа `Byte`, `Shortint` сдвигается поле из 8 бит, типа `Word`, `Integer` – 16 бит, `Longint` – 32 бита. Использовать N больше, чем эти величины или отрицательным бессмысленно – результатом будет 0.

Вещественные данные

Данные этого типа имеют своими значениями подмножество вещественных чисел, которые допустимы в компьютере. Они занимают в памяти от 4 до 10 байт и представляются в форме с плавающей точкой в 2 с/с. Количество значащих цифр числа ограничено, поэтому значения данных вещественного типа могут быть представлены в машине неточно.

Следующая таблица показывает, какие характеристики имеют вещественные данные. таблица 10):

Таблица 10 – Вещественные типы и их диапазоны.

Тип	**Диапазон модуля числа**	**Количество цифр мантиссы**	**Память в байтах**

Тип	Диапазон модуля числа	Количество цифр мантиссы	Память в байтах	
`Real`	$2.9*10^{-39}..1.7*10^{38}$	11-12	6	
`Single` (простой)	$1.5*10^{-45}..3.4*10^{38}$	7-8	4	Требуется установить режим компиляции 8087/80287 в меню Options подменю Compiler команда Numeric processing
`Double` (двойной)	$5.0*10^{-324}..1.7*10^{308}$	15-16	8	
`Extended` (расширенный)	$3.4*10^{-4932}..1.1*10^{4932}$	19-20	10	
`Comp` (целочисленные значения)	$0..\ 9.2*10^{+18}$	19-20	8	

Операции над вещественными данными

1) *Арифметические операции*: +, –, *, / (один из операндов может быть целый, но результат получится вещественного типа).

2) *Операции отношений*: =, >, <, <>, >=, <= (результат – **`true`** или **`false`**).

Замечание. *Надо помнить, что к операндам вещественного типа не стоит применять операцию отношения равно «=», поскольку условие может не выполниться из-за представления вещественных чисел в памяти ПК и неизбежных ошибок округления при подсчете значений выражений этого типа.*

Поэтому отношение $a1=a2$ лучше заменить отношением $abs(a1-a2)<E$, где E – некоторая достаточно малая величина – погрешность округления.

3) *Математические функции*: $abs(x)$, $sqr(x)$, $\sin(x)$, $\cos(x)$, $arctan(x)$, $ln(x)$, $exp(x)$, $sqrt(x)$.

В перечисленных функциях аргумент x может быть и данным целого типа, но результат, кроме $abs(x)$ и $sqr(x)$ – всегда вещественный.

В стандарте языка нет операции возведения в степень a^x (за исключением $sqr(x)$), поэтому для вещественного x используют тождество: $a^x = e^{x\ln a}$, но тут надо следить за знаком основания a, а для целочисленного x используйте соответствующий циклический алгоритм.

4) Другие функции:

$frac(x)$ – вычисление дробной части x (`fraction` – дробь);

$int(x)$ – вычисление целой части x, отбрасыванием дробной;

pi – возвращает значение числа π : 3.141592653897932385 (19 цифр);

random – генерирует значение вещественного случайного числа из диапазона [0..1) ;

$random(I)$ – генерирует значение целого неотрицательного случайного числа из диапазона [0, *I-1*];

randomize – изменение базы генерации случайных чисел. Применяется только один раз в программе.

Замечание. В языке нет некоторых встроенных функций, поэтому можно вычислять их, используя известные формулы:

$$\operatorname{tg} x = \frac{\sin x}{\cos x}, \qquad \operatorname{ctg} x = \frac{\cos x}{\sin x}, \qquad \arcsin x = \operatorname{arctg} \frac{x}{\sqrt{1-x^2}},$$

$$\arccos x = \frac{\pi}{2} - \arcsin x, \qquad \log(x) = \frac{\ln x}{\ln 2}, \qquad \log_{10}(x) = \lg(x) = \frac{\ln x}{\ln 10},$$

$$\operatorname{th} x = \frac{\operatorname{sh} x}{\operatorname{ch} x} = \frac{e^x - e^{-x}}{e^x + e^{-x}}, \qquad \operatorname{cth} x = \frac{\operatorname{ch} x}{\operatorname{sh} x} = \frac{e^x + e^{-x}}{e^x - e^{-x}}.$$

Выражения языка

Выражение – это формальное правило для подсчета некоторого (нового) значения. Понятие выражения присутствует практически в любом достаточно развитом языке программирования, а синтаксические правила для построения выражений очень схожи в разных языках.

В общем виде можно сказать, что выражения строятся из операндов, знаков операций и круглых скобок.

Операнды представляют собой «элементарные» значения. Ими могут быть переменные, константы, элементы массивов, вызовы функций и т. д.

Операции определяют действия по вычислению новых значений, исходя из значений операндов. Большинство операций являются бинарными, и знак операции записывается между операндами. Для унарных операций, а это @ – взять адрес, **not** – логическое отрицание, унарные ±, знак операции ставится перед операндом. Никогда нельзя опускать знак операции «*». Операции продуманы так, чтобы выражение писали в одну строку.

Приоритеты задают очередность выполнения операций в выражении. *Круглые скобки* ставятся тогда, когда нужно изменить порядок действий, который определен приоритетами операций. Количество открытых скобок должно равняться количеству закрытых. Часть выражения, заключенная в круглые скобки, при подсчетах рассматривается как отдельный операнд, т.е. все операции внутри подвыражения будут выполнены в первую очередь.

Операнд, находящийся между двух операций с разными приоритетами, связывается с операцией с большим приоритетом $(...+a*...)$, а если приоритеты равны, то берется слева.

Наибольший приоритет имеет операция вычисления значения функции, остальные – в соответствии с таблицей (таблица 11):

Таблица 11 – Приоритет операций.

Операция	**Приоритет**	**Вид операции**
`@, not, +, -`	Высокий, первый	Унарные
`*, /, div, mod, and, shl, shr`	Второй	Бинарные типа умножения

`+, -, or, xor`	Третий	Бинарные типа сложения
`=, <>, <, >, >=, <=, in`	Низкий четвертый	Отношения

Символьные данные

Переменные этого типа имеют описание `Char`, а значениями данных являются символы из множества `ASCII` (`American Standard Code for Interchange Information` – американский стандартный код для обмена информацией). Это множество состоит из 256 различных символов, причем первые 128 символов стандартные, остальные могут меняться. Во вторую часть альтернативной кодировки введен национальный алфавит (кириллица).

Первые 32 символа – управляющие, остальные имеют графическое представление. Порядковый номер (кодировку) значений символьного типа можно узнать из специальной литературы или распечатать. Эти значения занимают 1 байт.

Существуют и другие таблицы кодировки символов для персональных компьютеров, например кодировка под `Windows ANSI` (однобайтные символы), `UNICOD` (двухбайтные символы).

Если символьное значение имеет графическое представление, то оно выражается соответствующим знаком, заключенным в одинарные апострофы. Например: ‘*‘, ‘!‘,‘‘‘‘ (собственно апостроф в тексте удваивается) – это *первая форма представления символа*.

Если символ не имеет графического представления (это касается, например, символов управляющих, с кодами $0 \div 31_{10}$), тогда можно воспользоваться следующей эквивалентной формой записи символа: пишется символ # (диез) и целочисленный код символа (от 0 до 255). Например: #10 (перевод строки); #$A (то же в 16-й с/с); #13 (возврат каретки) – это *вторая форма представления символа*.

Кроме того, некоторые руководящие символы `ASCII` можно представить в следующей форме: ^C (нажимаем `Ctrl` и C), где C – условное обозначение управляющего символа. Первые 26 управляющих символов строятся, как ^ и буква от A до Z; номер буквы в алфавите соответствует номеру символа – *это третья форма представления символа*. Например, ^G = #7 (при выводе на печать будет звук) – символ с кодом 7, так как G – это седьмая буква латинского алфавита, ^J = #10, ^M = #13.

Символы псевдографики на обычной клавиатуре отсутствуют, но их можно получить так: при помощи клавиши `Alt` (нажать ее и удерживать) набрать код символа на дополнительной цифровой клавиатуре. Появится соответствующий символ.

Например, набирая `Alt` и 251, получим символ √.

Для значений символьного типа введены операции сравнения (фактически сравниваются коды символов как целые положительные числа).

Функции

Функция Ord(C) (Order – порядок), где C – типа Char, дает порядковый номер символа в таблице ASCII кодов.

Функция Chr(i) по порядковому номеру i дает символьное значение – символ. Если $0 \le i \le 255$, то этот символ можно найти в таблице кодов.

Имеет место Chr(Ord(C))=C и Ord(Chr(i))=i – следовательно эти функции взаимно обратные. Например, #32 = Char(32) – символ пробела.

Две функции Pred(c) и Succ(c) дают соответственно предыдущий и последующий символы. Их можно использовать и к другим простым дискретным типам. Не имеют смысла такие выражения: Succ(#255) и Pred(#0).

Функция UpCase переводит символы из нижнего регистра в верхний регистр для букв латинского алфавита: UpCase('a')⇒'A'.

Логические данные

Переменные этого типа имеют описание Boolean. Существуют два значения логического типа (правда и ложь), и эти значения в языке Pascal обозначаются стандартными идентификаторами **true** и **false**. Значения логических типов занимают 1 байт памяти. Внутреннее представление для **false** есть 0, для **true** – 1. Функция Ord возвращает значение 1 для логического значения **true** и значение 0 для логического значения **false**.

Над значениями логического типа возможны операции сравнения, причем считается, что **false** < **true**. Кроме того, существуют четыре стандартные логические операции: **and**, **or**, **xor**, **not** (таблица 12):

Таблица 12 – Таблица истинности операций and, or, xor, not.

and	true	false
true	true	false
false	false	false

or	true	false
true	true	true
false	true	false

xor	true	false
true	false	true
false	true	false

not
not true = false
not false = true

Значения логического типа можно применять для образования ветвлений.

Данные адресного типа

Язык Turbo Pascal имеет специальный адресный тип – Pointer (указатель). Значением адресного типа является адрес ячейки памяти, заданный по правилам операционной системы MS DOS.

Существует встроенная функция SizeOf(a), которая дает размер памяти аргумента в байтах. Например, вызов функции SizeOf(Pointer)дает 4 (байта).

Данные пользовательского типа

Существуют категории типов данных, вводимых программистами. Имя типа обозначается идентификатором. К ним в первую очередь относятся перечислимые и ограниченные (тип «диапазон») типы.

Объявляются новые типы в блоке описания типов **type** по следующему формату:

```
type
 Новый_тип_1   = массив_целых_чисел;
 Новый_тип_2   = множество_символов;
     ...
 Новый_тип_101 = целое_число; {тождественный тип}
 Новый_тип_102 = перечисленные_здесь_значения;
```

Стандартные типы не надо определять заново.

Данные перечислимого типа

Перечислимый тип – это тип данных, в которых количество всех возможных значений ограничено. Его можно расписать в ряд по значениям, перечисляя через запятую в круглых скобках названия (идентификаторы) элементов-значений типа (эти названия должны быть уникальными в пределах программы). Перечислимому типу можно присвоить имя, тогда он описывается в секции **type**.

Формат именованного перечислимого типа:

```
type
  имя типа=(идентификатор_1,... идентификатор_n);
var
 идентификатор_пер_1, идентификатор_пер_2,...  :
 имя типа;
```

Формат неименованного перечислимого типа:

```
 var идентификатор_пер_1, идентификатор_пер_2,...:
       (идентификатор_1,... идентификатор_n);
```

Переменная такого типа сможет содержать те значения, которые указаны в его перечислении, иначе возникает ошибка. Максимальное число элементов в одном перечислении равно 65535. Сохраняется значение перечислимого типа в беззнаковом байте, если перечислено не более 256 значений, и как беззнаковые слова – в остальных случаях.

Любой перечислимый тип имеет внутреннюю нумерацию значений от 0 до последнего. Номер каждого значения можно получить функцией `Ord(x)`, которая возвращает целое число в форме `Longint`. Например, `Ord(NufNuf)` возвращает значение `1`.

Имеет место обратная запись: `P := Personages(i);` где $0 \le i \le 2$, `i` – целое. Например, при `i=2`, переменная `P` получает значение `NafNaf`. В переменную P записывается значение, соответствующее заданному порядковому номеру элемента перечисления. К переменным перечислимого типа применимы функции `Succ(x)` и `Pred(x)`.

Поскольку значения перечислимого типа упорядочены, их можно сравнивать: кто имеет больший порядковый номер, тот и больше. Например, результат сравнения `NufNuf<NafNaf` есть **`true.`**

Недостатки перечислимого типа: значения данных перечислимого типа не могут быть выведены на экран, или принтер, или в файл и не могут быть введены с клавиатуры, так как это внутренние данные (но решить эти проблемы можно, заведя, например, массив строк с такими же названиями, а индексы массива – это те же имена данных перечислимого типа).

Данные интервального типа

Для введения нового типа-диапазона в блоке определения типов **`type`** нужно указать имя этого типа и пределы диапазона через две точки:

```
const      n=10;
type       Radius     =  1..90;
           Capsletter = 'A'..'Я';
           Diap       = 2*n..3*n;
```

Границы могут быть и выражениями, но нельзя, чтобы выражение начиналось со скобки, так как скобка – это отметка начала перечисления.

Компилятор при каждой операции с переменной интервального типа может генерировать подпрограммы проверки, чтобы установить, остается ли значение переменной среди диапазона ее значений. Для этого нужно установить директиву компилятора {$R+} (по умолчанию {$R–}).

С данными интервального типа могут работать функции `Succ, Pred, Ord`.

К ним можно применять все стандартные подпрограммы и операции, которые определены для базового типа. Только если диапазон взят из базового типа, который может выводиться на экран или вводится с клавиатуры (или ввод-вывод в файл), тогда и ограниченные данные можно таким образом вводить-выводить.

1.7. Простейшее определение процедур и функций. Файловая система

Содержание

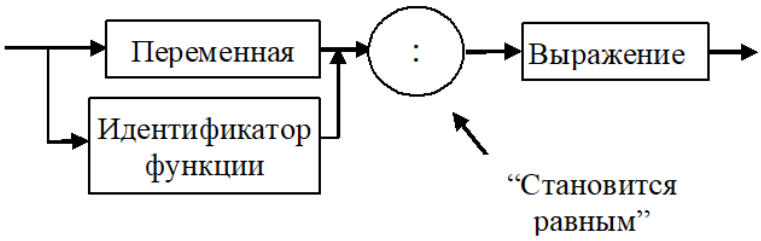
- Простейшее определение процедур и функций.
- Параметры.
- О файловой системе:
  - файловый тип,
  - стандартные текстовые файлы.

Присваивание значений

Оператор присваивания значения выполняет вычисление значения, определяемого выражением в правом операнде, и сохраняет вычисленное значение в левом операнде. В левом операнде может использоваться переменная или идентификатор функции. Если в левом операнде указан идентификатор функции, то это означает, что оператор выполняется в описании

функции и, в этом случае, оператор присваивания значений устанавливает значение, которое будет возвращаться функцией в точку вызова.

Общий вид оператора присваивания значения:



Порядок выполнения оператора присваивания значения:

- подсчитывается значение выражения;
- если тип выражения «*совместим по присваиванию значения*» с типом переменной, тогда значение переменной изменяется на подсчитанное;
- если совмещения нет, тогда на этапе компиляции программы появляется сообщение компилятора о возникшей ошибке.

Присваивание значений допускается для данных всех типов – простых и составных, кроме файлового типа. Отметим важные моменты:

При присвоении значения данному простого типа нужно помнить следующее:

•тип выражения не должен выходить из диапазона возможных значений данного типа переменной, он может являться подмножеством типа переменной;

•данным вещественного типа можно присваивать значения целочисленного типа;

•данным целого типа нельзя присвоить значение вещественного типа. Надо пользоваться функциями «округлить и представить как целое» – `round` и `trunc`, которые возвращают значение типа `Longint`.

2) *При присваивании значения данному составного типа нужно помнить следующее*: типы в левом и правом операндах оператора присваивания должны быть эквивалентными или тождественными.

```
type
        Type2 =   …
        Type1 = Type2;   {Это эквивалентно }
var
        A, B : Type1;  {тождественные типы для A и B}
        C    : Type1;
        D    : Type2;  {тождественные типы для C и D}
```

Простейшее определение процедур и функций

Процедура и функция – это средства, которые позволяют многократно использовать в разных местах программы один раз описанный фрагмент алгоритма.

Подпрограммой называется именованная группа операторов языка, которую можно вызвать для выполнения через имя любое количество раз из разных мест программы.

В языке Pascal для организации подпрограмм используются *процедуры* и *функции*.

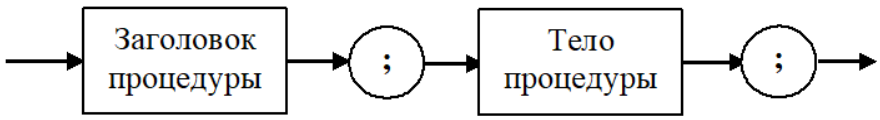
Процедура – это независимая именованная часть программы, предназначенная для выполнения определенных действий. По структуре она напоминает программу в миниатюре.

После однократного описания процедуру можно вызвать в других частях программы по имени. Когда завершится выполнение процедуры, то продолжится выполнение программа с оператора, который написан непосредственно за оператором вызова процедуры (если в процедуре не был выполнен оператор останова). Имя процедуры не может находиться в выражении в качестве операнда. Упоминание этого имени в тексте программы приводит к активизации процедуры и называется ее вызовом.

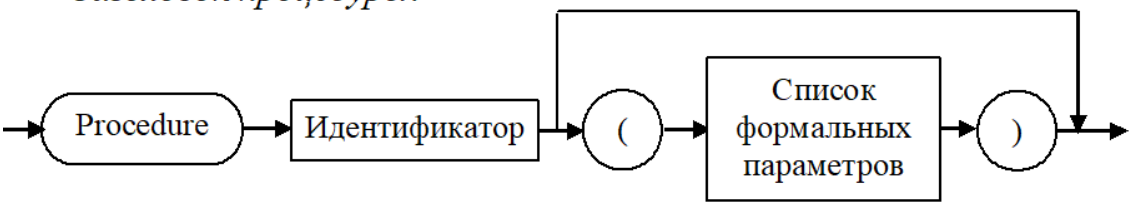
Функция аналогична процедуре, но:

- функция возвращает в точку вызова результат своей работы;
- имя функции может входить как операнд в выражение.

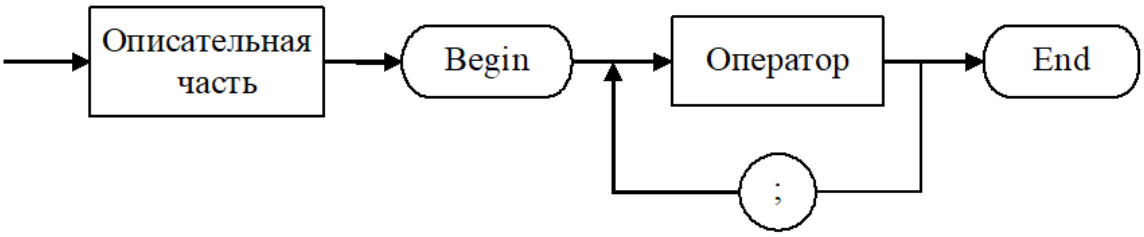
Синтаксическая диаграмма простейшего определения процедуры:



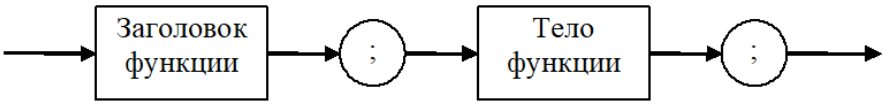
Заголовок процедуры:



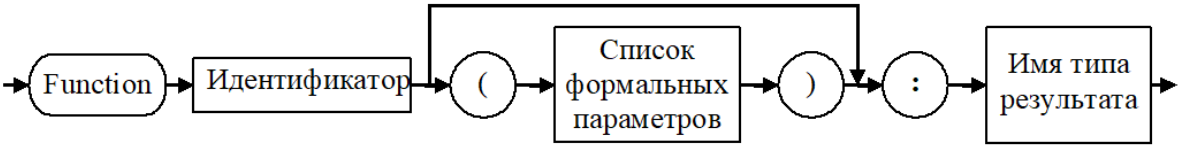
Тело процедуры в простейшем виде определяется следующей синтаксической диаграммой:



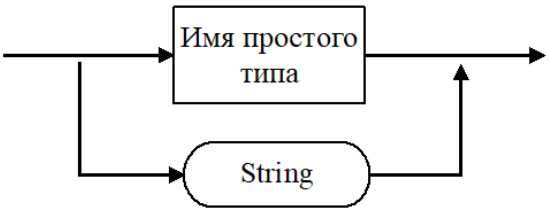
Синтаксическая диаграмма простейшего определения функции:



Заголовок функции:



Имя типа результата или тип функции:



Тело функции напоминает тело процедуры, однако среди операторов тела функции должен быть хотя бы один оператор, который имени функции надает значение результата работы функции.

Пусть надо вычислить $y = f(x), \quad f(x) = e^{x^2} \cos(x)$.

Пример простейшего определения процедуры:

```
procedure FP(x : Real; var y : Real);
begin
     y:=exp(Sqr(x)) * Cos(x)
                                {выражение f(x)}
end;
```

В дальнейшем вызов процедуры, например, такой

```
FP(x, y).
```

Пример простейшего определения функции:

```
function F(x : Real) : Real;
begin
     F := exp(Sqr(x)) * Cos(x)
                                {выражение f(x)}
end;
```

В дальнейшем вызов функции может быть таким:

```
Y := F(x).
```

Параметры

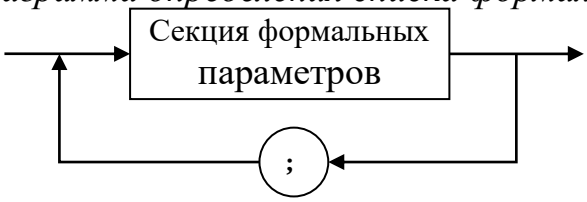
Параметры обеспечивают механизм, позволяющий выполнять подпрограммы с различными данными.

Процедуры и функции могут иметь (или не иметь) список формальных параметров. Каждый параметр описывается некоторым именем. Тип же параметра может быть любым из разрешенных.

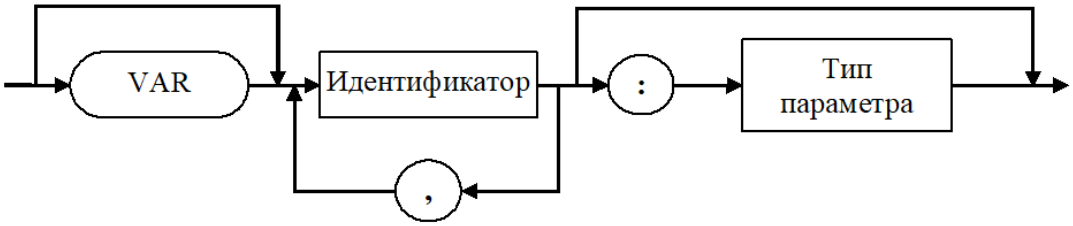
При вызове подпрограммы формальные параметры заменяются фактическими параметрами – фактическими данными. Списки формальных и фактических параметров должны иметь одинаковое количество параметров и быть эквивалентными (иначе компилятор заметит ошибку: «**type mismatch**» – несовпадение или несоответствие типов).

Параметры перечисляются в заголовке подпрограммы в списке параметров.

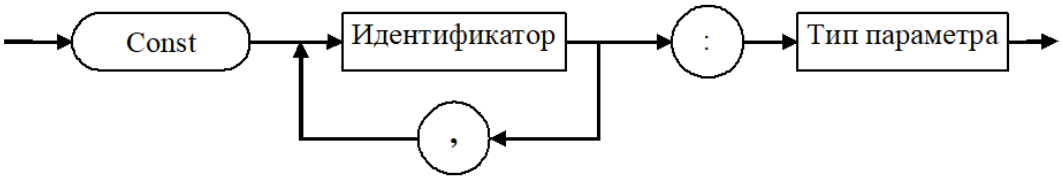
Синтаксическая диаграмма определения списка формальных параметров:



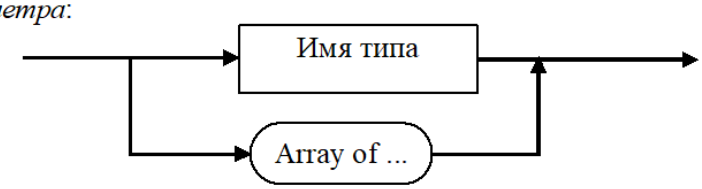
Секция формальных параметров



или же



Тип параметра:



Из этих диаграмм следует, что возможны следующие виды параметров:

параметры-значения с типом (без `var`)

```
идентификатор : тип_параметра;
```

параметры-переменные (с `var`) с типом

```
var идентификатор : тип_параметра;
```

параметры-переменные (с `var`) *без типа*

```
var идентификатор;
```

В подпрограмме любые изменения параметров, описанных в секции **`const`**, не допускаются. Компилятор строит программу так, что фактические параметры обрабатываются на своем месте в памяти (при помощи ссылок). Значит, при активизации (вызове) подпрограммы на месте фактических параметров-констант передаются адреса их нахождения в памяти.

Конструкция

```
array of тип_элементов
```

задает так называемый *открытый массив*.

О файловой системе

Файловый тип

Определяя переменные файлового типа в программе, появляется возможность общения с периферией ПК и в том числе накапливать данные, чтобы позже обращаться к ним. Файловые типы языка Pascal отличаются только типами данных, содержащихся в них.

Текстовые файлы – это файлы, которые состоят из ASCII кодов символов (вместе с управляющими кодами). Они организуются по строкам, содержат коды #13 = ^M и #10 = ^J и обязательно специальный код, который называется концом файла (код #26 = ^Z). Основное свойство таких файлов в том, что они могут быть созданы и *программным путем*, и в каком-то *текстовом редакторе*. В программе описываются служебным словом `Text`.

Файлы с типом – компонентные файлы. В отличие от текстовых файлов, файлы с типом строятся из машинных представлений данных заранее объявленного типа. Они хранят данные в том же виде, что и в оперативной памяти ПК. Основное свойство таких файлов в том, что они не могут быть созданы непрограммным путем. Такие файлы имеют дело с данными наперед объявленного типа.

Нетипизированные файлы имеют дело с произвольными наборами байтов независимо от их структуры и природы.

При помощи файловых типов определяют файловые переменные, которые имеют одно свойство: они не могут участвовать в операторах присваивания значения.

Стандартные текстовые файлы

В языке Pascal существуют два стандартных текстовых файла: `Input` и `Output`.

Существуют четыре процедуры, при помощи которых совершаются операции ввода-вывода данных, связанные с файлами типа `Text`:

$$\left.\begin{matrix} read \\ readln \end{matrix}\right\} ввод \qquad \left.\begin{matrix} write \\ writeln \end{matrix}\right\} вывод$$

Информация идет с клавиатуры в стандартный файл `Input`, стандартный файл `Output` связан с дисплеем.

Процедуры ввода-вывода могут быть как с параметрами, так и без них. Если вводим (или выводим) несколько данных, они отделяются в списке ввода (вывода) оператора запятой.

Описание текстового файла:

```
var Ft : Text;
```

Форматы процедур для чтения (ввода) информации из текстового файла:

```
Read;
Read(Ft, x);
Read(Ft, x1, …, xn);
Readln;
Readln(Ft, x);
Readln(Ft, x1, …, xn);
```

где первый параметр `Ft` – имя файловой переменной типа `Text`, второй параметр, если он есть, и все последующие – переменные разрешенного типа, в которые вводятся какие-то значения.

Если `Ft = Input`, тогда в операторах ввода имя файла может быть опущено:

`Read(Input, x)` ⇔ `Read(x)`.

Если `Read` или `Readln` – процедуры без параметра, тогда ожидается любой ввод с клавиатуры, т.е. пауза.

Оператор `Read(Ft, x1, …, xn);` равносилен последовательности операторов

```
begin  Read(Ft, x1); …; Read(Ft, xn) end;
```

Оператор `Readln(Ft, x1, …, xn);` равносилен последовательности операторов

```
begin  Read(Ft, x1); …; Readln (Ft, xn); end;
```

где `x1`, …, `xn` – переменные допустимых типов.

Форматы процедур для вывода информации в текстовый файл:

```
Write(Ft, x);
Write(x);
```

Оператор `Write(Ft, x1, …, xn);`
равносилен последовательности операторов

```
begin Write(Ft, x1); …; Write(Ft, xn); end;
```

```
Writeln(Ft, x);
Writeln(x);
```

Оператор `Writeln(Ft, x1, …, xn);`
равносилен последовательности операторов

```
begin Write(Ft, x1); …; Writeln(Ft, xn); end;
```

где первый параметр `Ft` – имя файловой переменной типа `Text`.

Если `Ft = Output`, тогда его можно опустить, т.е.

`Write (Output, x)` ⇔ `Write (x)`.

Например,

```
Writeln(Ft){выводит пустую строку в файл Ft    }
Writeln    {выводит пустую строку в файл Output}
```

Замечание. Значения, которые выводятся, автоматически переводятся в символьное представление, а при вводе – переводятся из символьного представления во внутреннее в соответствии с их типом.

1.8. Базовые операторы Pascal

Содержание

- Ввод данных разных типов.
- Вывод данных разных типов.
- Простые операторы:
  - пустой оператор,
  - оператор безусловного перехода `goto`,
  - оператор вызова процедуры,
  - составной оператор.

Ввод данных разных типов

Тип переменных при вводе с клавиатуры может быть только простым, то есть целым, вещественным, символьным или строковым, или совместимыми с ними (диапазоны).

Когда вводятся *числовые значения*, то они должны набираться по правилам синтаксиса и быть разделенными. Два числа считаются разделенными, когда между ними есть хотя бы один пробел, или символ(ы) табуляции (#9), или конец строки (#13#10).

Когда вводится *символьное* значение, тогда в соответствующую переменную запишется очередной символ за последним символом, введенным до этого.

Ввод строк. Строки начинается сразу за последним, введенным до этого символом (с первой позиции, когда строковая переменная стоит первой в списке ввода). Читается количество символов, равное объявленной длине строки. Но когда во время чтения попался символ #13, тогда чтение строки останавливается, но сам символ #13 в строковую переменную не записывается, ведь он служит разделителем строк.

Ввод *логических значений.* Так как это внутренний тип и не приспособлен к вводу, то можно применить следующую хитрость:

```
var  by  : Byte;
     boo : Boolean absolute by;
  {позволяет переменную boo разместить на место
   переменной by и ввести значение в by}
      . . .
     Read(by);
                    {закодируйте 0 = false; 1 = true
                    и введите байт 0 или 1 – это и
                    будет считаться логическим значением}
```

После набора данных для одного `Read` нажимается клавиша ввода <Enter>. Значения переменных должны вводиться в строгом соответствии с порядком перечисления их в списке ввода и с синтаксисом языка. Когда же, например, `x1` имеет тип `Integer`, а при вводе набирается значение, отличное от цифры, то возникнет ошибка ввода-вывода. Сообщение об ошибке имеет вид: `I/O error XX`,

где XX – код ошибки. Пояснительный текст поможет определить причину программного прерывания.

Процедура ввода Readln аналогична Read, за исключением того, что после считывания последнего в списке значения для последней в списке ввода переменной происходит переход курсора в следующую строку.

Фактически информация идет в буфер ввода, а затем отображается на экране.

Когда идет чтение с Input или из текстового файла, который назначается на экран, вводится одна строка входного текста за одну операцию. Строка запоминается во внутреннем буфере текстового файла и когда строка читается, этот буфер используется как входной источник.

Для ввода массивов необходимо организовать их поэлементный ввод, записи вводятся по полям. Ввод множеств напрямую просто не предусмотрен.

Вывод данных разных типов

Процедура записи Write выполняет вывод данных в строку. Если вывод строки достигает правой границы экрана, строка разрывается и выполняется переход на следующую строку, а если это последняя строка экрана, то происходит сдвиг (прокрутка) экрана. Вывод может идти в бесформатном или форматном вариантах. В операторах вывода

```
Write(y1,…, yn);
Writeln(y1, …, yn);
```

yi (1≤i≤n) может иметь три варианта записи:

E (бесформатный вывод)

E:F (форматный вывод)

E:F:d (форматный вывод только для данных вещественного типа!).

Здесь E, F, d – некоторые выражения, E – выводимое выражение, (в простом случае – имя переменной разрешенного типа). Целочисленное выражение F указывает, сколько позиций может занять E (общая ширина поля вывода). Целочисленное выражение d указывает, сколько позиций в поле F занимает дробная часть. Для отображения очередного значения E при бесформатном выводе на экране или принтере используется поле (заданное системой по умолчанию), величина которого зависит от типа выражения E. Выражение E принадлежит к одному из следующих типов: целочисленный, вещественный, символьный, строковый, логический. При выводе массивов необходимо организовать их поэлементный вывод, записи выводятся по полям. Вывод множеств напрямую не предусмотрен.

Вывод символов

В бесформатном варианте вывода символ занимает одну очередную позицию.

В форматном варианте вывода формат задается конструкцией E:F, где E – выводимое выражение, F – целочисленное выражение. Если F – целое положительное значение, то информация прижимается к правому краю поля длины F. Когда F – целое отрицательное значение, тогда информация

прижимается к левому краю поля длины F и потом игнорируются остальные позиции.

Вывод строковых данных

Значение строковых данных при бесформатном выводе отображается без ограничивающих их апострофов.

Когда на дисплее при выводе данное достигло правой стороны экрана, строка продолжается без перерыва в следующей строке экрана с первой позиции.

Форматный вывод имеет такой же смысл, как и при выводе символа: задаем ширину поля вывода конструкцией E:F, где E – то строковое выражение, что выводим, F – целочисленное выражение – ширина поля. Если ширины поля не хватает для длины строки – формат игнорируется. Если целочисленное выражение F – положительно и величина его больше текущей длины строки – данное прижимается к правой стороне, если же отрицательно – к левой.

Вывод логических значений

В текстовый файл попадает информация: строка **true** или **false**. Форматный вывод задается так же, как и для строки символов.

Вывод целочисленных значений

Положительное число выводится без знака, отрицательное – со знаком минус ”–“. Количество позиций в бесформатном варианте вывода ограничивается величиной данного и занимает равно столько места, сколько в значении данного десятичных цифр.

В форматном варианте задается конструкция E:F, где E – то целочисленное выражение, что выводим, F – целочисленное выражение, которое регулирует ширину поля вывода как и для строки символов.

Если данное не вмещается в отведенное поле, тогда F игнорируется и данное займет столько позиций, сколько требует его значение.

Вывод данных вещественного типа

В бесформатном выводе данные печатаются в форме с плавающей точкой по шаблону, который задается по умолчанию. По умолчанию значение типа Real занимает 17 позиций, и выводится по схемам:

$$-X.\underbrace{XXXXXXXXXX}_{10}E\pm XX \quad \text{или} \quad \square X.\underbrace{XXXXXXXXXX}_{10}E\pm XX$$

Здесь X – какая-то десятичная цифра.

Форматный вывод имеет конструкцию E:F:d, где E – выводимое выражение (простой случай – имя переменной вещественного типа), F – общая длина поля, d – количество цифр в мантиссе после точки:

$$\overbrace{\pm X\ldots X.\underbrace{X\ldots X}_{d}}^{F} \qquad F \geq d+3.$$

Когда применяется форматный вывод вещественных значений в форме с плавающей точкой, надо предусматривать следующие позиции:

а) Когда данное типа Real:

$$-X.X\ \ldots\ XE\pm XX$$

```
                ↑↑↑↑     ↑↑↑↑
```

б) Когда данное типа `Single, Double, Extended`:

```
              -X.X … XE±XXXX
              ↑↑↑↑     ↑↑↑↑↑↑
```

Отмеченные символом ↑ позиции обязательно нужно предусмотреть при подсчете длины поля `F`.

Когда `d=0`, выводится целое округленное значение числа без точки. Когда `d>24`, тогда `d` игнорируется и вывод происходит в форме с плавающей точкой. Когда заданная целочисленная величина `F` не достаточна для вывода данного, тогда вместо `F` берется необходимая величина, `d` – остается прежним. Когда `:d` – отсутствует, тогда вывод происходит в форме с плавающей точкой в поле шириною `F` позиций.

Простые операторы

К простым операторам относятся операторы, которые описываются простыми конструкциями.

Пустой оператор

Пустой оператор не содержит никаких символов и не выполняет никаких действий. Пример: ; ; ; – два пустых оператора; M:; – помеченный пустой оператор.

Это оператор-невидимка. Он может быть расположен в любом месте программы, где синтаксис языка позволяет присутствие оператора.

Скорее всего, мы его применяем случайно, когда ставим лишние «;».

Оператор безусловного перехода goto

Оператор **`goto`** `M` передает управление на оператор с меткой `M` и этим изменяет последовательность линейного выполнения программы.

Оператором **`goto`** нельзя передавать управление из программы в подпрограмму и наоборот.

Использование оператора **`goto`** нарушает наглядность программы и затрудняет ее анализ, отладку и модификацию. При структурном программировании можно так построить алгоритм, что не будет безусловной передачи управления.

Введено много других операторов, чтобы не использовать **`goto`**:

`exit` – позволяет завершить работу текущего программного блока;

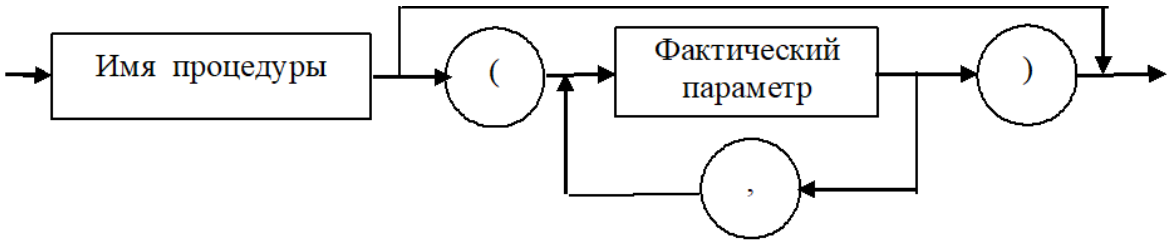
`halt` – позволяет завершить работу программы;

`break` – позволяет досрочно закончить цикл;

`continue` – позволяет перейти на продолжение цикла.

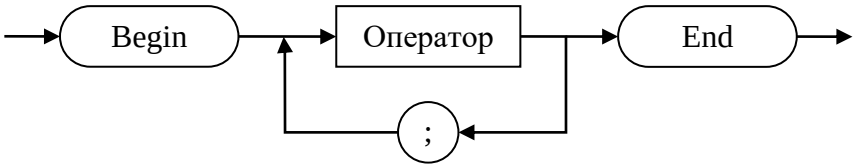
Оператор вызова процедуры

Оператор вызова процедуры не имеет никакого служебного слова. Служит для активизации предварительно определенной процедуры.



Составной оператор

Довольно часто в программе возникает необходимость объединить группу операторов в один оператор. Подобно тому, как любое выражение, взятое в скобки, становится операндом, составной оператор образуется из последовательности операторов, заключенных в скобки, но не круглые, а специальные операторные скобки – **`begin`** и **`end`**. Между этими скобками можно написать любое число операторов, которые отделяются друг от друга «;» – точкой с запятой (так как «;» ставится между операторами, то перед скобкой **`end`** точка с запятой фактически не нужна, однако если она ставится, то здесь просто появляется пустой оператор).



Составной оператор воспринимается как единое целое и может находиться в любом месте программы, где синтаксис языка допускает наличие оператора.

Обычно составной оператор используется как тело циклов, или при написании условных операторов.

1.9. Структурные операторы

Содержание

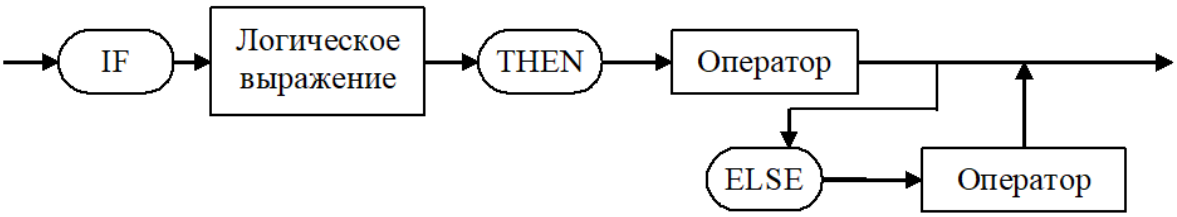
- Операторы ветвления:
  - условный оператор `if`,
  - оператор выбора `case`.
- Операторы повторения:
  - оператор повторения `for`,
  - оператор повторения `repeat`,
  - оператор повторения `while`.
- Быстрая степень.

Операторы ветвления

Условный оператор и оператор выбора позволяют разделить вычислительный процесс на две или более ветвей, выполняя каждый раз (в зависимости от условия) требуемый оператор.

Условный оператор

Условный оператор имеет вид:

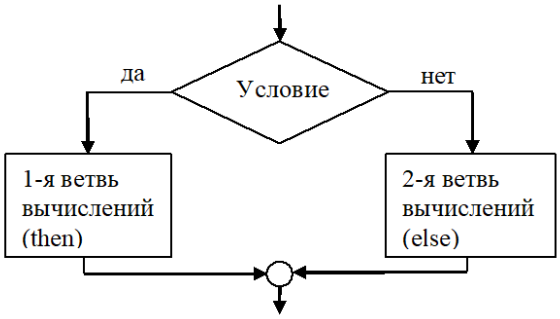


Оператор **if** работает так: вычисляется значение логического выражения и, если оно истинно, выполняется оператор, стоящий после **then**, если ложное – оператор, стоящий после **else**, если же часть с **else** отсутствует, то в случае ложного значения логического выражения никаких действий не выполняется. Далее выполняется следующий оператор, если в **if** не было оператора **goto** или операторов выхода (**Halt**, **Exit**).

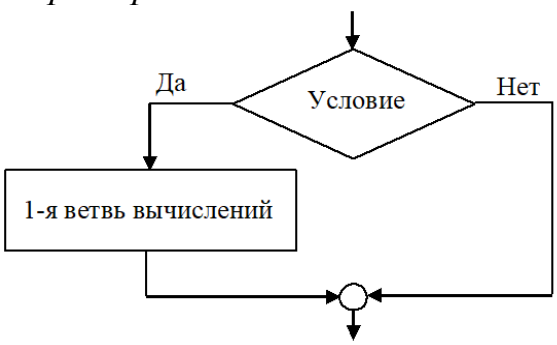
Методы и приемы программирования

Схематично разветвляющийся процесс может быть отображен при помощи структуры выбора.

Полную форму оператора **if** можно отобразить следующей блок-схемой:



Неполная форма оператора **if**:



Формат оператора **if**:

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

Сокращенная форма оператора **if**.

```
if условие then оператор;
```

Один оператор **if** может входить в состав другого **if**, в этом случае говорят о вложении операторов.

```
if  условие1  then
    if  условие2  then  оператор1
    else  оператор2
else  оператор3;
```

При вложении операторов каждое **else** соответствует тому **then**, которое непосредственно ему предшествует.

Замечание. Оператор, стоящий после **then** или **else** может быть или простым, или же составным, то есть иметь вид **begin ... end**.

Если при реализации вложенных условных операторов использовать их полную и неполную формы записи, то легко допустить ошибку. Можно вместо неполной формы оператора **if** применять полную форму с использованием пустого оператора после **else**, но лучше переформулировать логическое выражение. Приведем примеры.

1. Избегайте такой конструкции:

```
if (a>=b) and (mark>0)  then {пустой оператор}
    else  if (a<-5.5) or (b>7.5)  then y := 0
              else {пустой оператор}
```

1.1. Лучше записать

```
if not((a>=b) and (mark>0)) and((a<-5.5) or (b>7.5))
   then y := 0;
```

1.2. Еще лучше ввести логическую переменную d, выполнить оператор

```
d:= ((a>=b) and (mark>0)) and((a<-5.5) or (b>7.5));
```

а затем

```
if d then y := 0;
```

2. Укажите синтаксическую ошибку в следующем операторе.

```
if  x > y  then
       begin
        A := B;
        C := D;
       end;
        else A:=0;
```

Транслятор покажет на **else** и выдаст сообщение об ошибке. Нужно помнить, что перед **else** точка с запятой не ставится.

3. Синтаксически правильной, но бессмысленной, является следующая конструкция (транслятор на ошибку не покажет).

```
if not P then;
  begin
```

```
 C:=Cos(z);
 D:=x*y-Sqrt(z);
end;
```

Наверное, после **then** ”;“ лишняя, потому что присвоение значений переменным C и D будет выполнено независимо от значения P.

4. Вместо условного оператора

```
if a > b then p := true else p := false;
```

проще и эффективнее использовать оператор присваивания. `p := a > b`.

5. Программируя вложенные условные операторы, нужно помнить, что условия в них проверяются последовательно, и поэтому первым надо писать условие, вероятность выполнения которого наибольшая. Например, нужно ловить попадание точки `x` в интервалы (0,1], (1,10], (10,100], (100,500]. Если все значения `x` равновероятны, то, поскольку длина последнего интервала самая большая, то вероятность попадания `x` в него самая большая. Получим такой фрагмент программы

***Вариант* 1**:

```
if (100 < x) and (x <= 500) then p := 1
else
    if (10 < x) and (x <= 100) then p := 2
    else
        if (1 < x) and (x <= 10)  then p := 3
        else
            if (0 < x) and (x <= 1) then p := 4
                                    else p := 0;
```

Однако, следующий вариант 2 более эффективен, потому что содержит меньше проверок.

***Вариант* 2**:

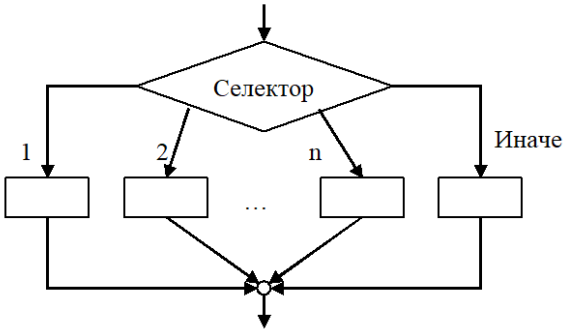
```
if  x > 500  then p := 0
else
     if x > 100 then p := 1
     else
         if x > 10 then p := 2
         else
              if  x> 1 then p := 3
              else
                      if x > 0 then p := 4
                               else p := 0;
```

Оператор выбора

Оператор выбора является некоторым обобщением оператора **if** и позволяет сделать выбор из произвольного числа имеющихся вариантов (ветвей). Этим осуществить ветвление по многим направлениям.

Рассмотрим, как используется оператор выбора сначала на схеме. Допустим, нужно реализовать процесс, который имеет такой вид: идет разветвление по одному из *n* путей в зависимости от какого-то значения «выражения-селектора» (или переключателя) порядкового типа.

Схема оператора **case**:



Формат оператора **case**:

```
case выражение-селектор of
    Список_1_констант_выбора: оператор_1;
    Список_2_констант_выбора: оператор_2;
          ...
    Список_n_констант_выбора: оператор_n;
else операторы
end;
```

Ветвь **else** может отсутствовать. Обратите внимание, что перед служебным словом **else** может стоять символ «;», но он может быть и опущен.

Сначала вычисляется значение выражения-селектора, затем обеспечивается реализация того оператора, константа выбора которого равна текущему значению селектора. Когда же ни одна из констант выбора не ровна текущему значению селектора, тогда:

- если есть вариант **else**, то выполняются операторы, записанные за ним;
- если отсутствует **else**, то выполняется первый оператор за пределами **case**.

Селектор должен относиться к одному из *простых* перечислимых типов: целочисленных (в диапазоне –32768 .. 65535), логическому, символьному или пользовательскому типов (`Longint`, `Real`, `String` – не разрешается).

Список констант выбора состоит из произвольного количества значений или диапазонов, которые отделяются друг от друга запятыми.

Примеры:

1. Ниже записан оператор с использованием списка констант выбора интервального типа, `I` – целое:

```
case I of
```

```
   1..10: Writeln('число', I:4, ' в диапазоне 1-10');
  11..20: Writeln('число', I:4, ' в диапазоне 11-20');
  21..30: Writeln('число', I:4, ' в диапазоне 21-30');
else    Writeln('число',I:4,' за границами контроля');
end;
```

2. Ниже записан оператор с использованием списка констант выбора целочисленного типа, `i` – целое:

```
case i of
0, 1:  x:=0;
   2:  x:=Sin(x);
   3:  x:=Cos(x);
   4:  x:=exp(x);
   5:  x:=ln(x)
end;
```

3. Ниже записан оператор с использованием списка констант выбора перечислимого пользовательского типа:

```
Var
      season: (Winter, Summer, Spring, Autumn);
begin       …
case season of
     Winter : Write ('Зима');
     Summer : Write ('Лето');
     Spring : Write ('Весна');
     Autumn : Write ('Осень')
  else   Writeln('Период не определен')
end;
```

Из этого примера делаем вывод: аналогичным образом можно организовать ввод и вывод данных перечислимого типа и обойти соответствующие ограничения языка по вводу данных перечислимого типа.

4. Ниже записан оператор с использованием списка констант выбора символьного типа:

```
var
    ch: Char; x, y, z: Integer;
        …
case ch of
     '+' : z:=x+y;
     '-' : z:=x-y;
     '*' : begin z:=x*y; Writeln(x,y,z); end;
     '/' : begin z:=x div y; Writeln(x,y,z); end;
     '^' : Writeln(x, '^', y, '=?');
end;
```

5. Оператор **`if`** в интерпретации оператора **`case`**:

```
var
```

```
   L: Boolean;
        …
case L of
     false : begin … Write('ложь'); … end;
     true  : begin … Write('истина'); … end
end;
```

6. Сложный выбор в интерпретации оператора **case**:

```
case i of
     1, 3, 5, 7, 9 : Write('нечетная цифра');
     0, 2, 4, 6, 8 : Write('четная цифра');
     10..100       : Write('от 10 до 100')
  else Write('<0 или >100');
end;
```

Замечание: Список констант выбора напоминает метку, но это не метка, на нее нельзя ссылаться в операторе перехода. Ее не нужно объявлять в **label**.

Примеры программ

Задача 1. Пусть дано натуральное число $N < 100$. Написать программу, которая выводит запись этого числа в римской системе счисления.

Решение. Для записи чисел первой сотни используются буквы I, V, X, L, C (1, 5, 10, 50, 100). Так как символы, означающие количество десятков, и символы, которые означают количество единиц, могут быть выведены независимо друг от друга, то в программе можно для вывода тех или других использовать оператор выбора.

```
program RIMSKIE;
 var     N: Integer;
begin
Read(N);
if (N<1) or (N>99) then
   Writeln('неправильно задано исходное число N=', N)
   else
   begin
   Writeln(N, ' арабскими цифрами');
  {выбор по количеству десятков}
 case N div 10 of
       1: Write('X');
       2: Write('XX');
       3: Write('XXX');
       4: Write('XL');
       5: Write('L');
       6: Write('LX');
       7: Write('LXX');
       8: Write('LXXX');
       9: Write('XC');
```

```
        end;
   {выбор по количеству единиц}
  case N mod 10 of
        1: Write('I');
        2: Write('II');
        3: Write('III');
        4: Write('IV');
        5: Write('V');
        6: Write('VI');
        7: Write('VII');
        8: Write('VIII');
        9: Write('IX');
        end;
   Write(' – римскими цифрами')
   end;
   Readln;
end.
```

Задача 2. Задано число, которое корректно записано двумя римскими цифрами. Вывести его значение, используя арабские цифры.

Решение. Будем считать, что число задано двумя латинскими буквами из множества (I, V, X, L, C, D, M). Соответственно римской системе записи чисел, если символ, который обозначает меньшее число, записан перед символом, обозначающим большее число, то меньшее число вычитается из большего, иначе числа складываются.

```
program ARABSKIE;
var
    RIMS1, RIMS2: Char;                    {римские цифры}
    ARAB1, ARAB2: Integer;                 {арабские цифры}
begin
 Readln(rims1, rims2);
        { задаем с клавиатуры символы без апострофа!}
 Writeln(rims1, rims2, ' – римскими цифрами');
 case rims1 of
     'I': arab1:=1;
     'V': arab1:=5;
     'X': arab1:=10;
     'L': arab1:=50;
     'C': arab1:=100;
     'D': arab1:=500;
     'M': arab1:=1000
  else
         Write('Ошибка'); Exit;
end;
case rims2 of
     'I': arab2:=1;
```

```
        'V': arab2:=5;
        'X': arab2:=10;
        'L': arab2:=50;
        'C': arab2:=100;
        'D': arab2:=500;
        'M': arab2:=1000
    else Write('Ошибка'); Exit;
    end;
    if arab1<arab2 then
                    Write(arab2-arab1)
                 else
                    Write(arab2+arab1);
    Write(' – арабскими цифрами')
end.
```

Если в качестве исходных данных будут введены не римские цифры, то ошибка будет определена соответствующим оператором выбора и напечатано слово `Ошибка`.

Операторы повторения

Поскольку цикличность – наиболее характерная особенность значительной части практически используемых алгоритмов, вопрос программирования циклов является одним из важных.

При циклических процессах типичным является случай, когда момент окончания циклического процесса определяется текущим значением некоторой переменной, которую называют *параметром цикла*.

Для гибкого управления циклическими операторами **`for`**, **`while`**, **`repeat`** существуют две процедуры – **`break`** и **`continue`**.

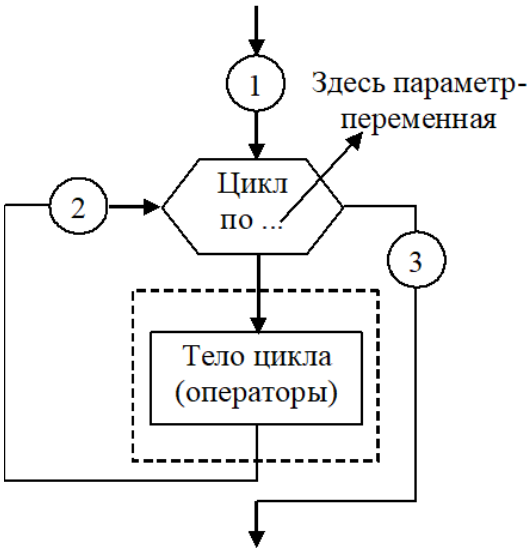
`break` – реализует “сразу же” выход из цикла;

`continue` – обеспечивает досрочное завершение очередного прохода цикла. Это фактически эквивалент передачи управления в конец тела цикла циклического оператора.

Введение этих операторов практически исключает потребность в операторе **`goto`**.

Если количество повторений известно заранее – или во время составления программы, или к моменту входа в цикл, тогда лучше использовать оператор цикла с параметром (схема а).

а) `for … to … do, for … downto … do`



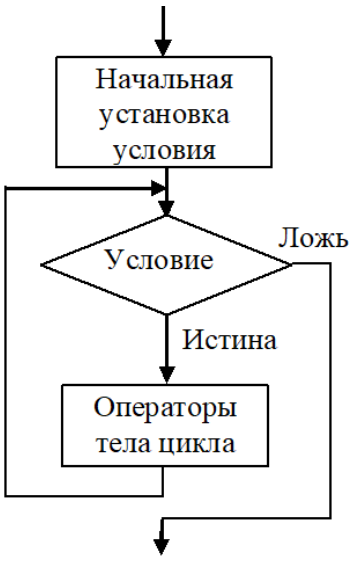
1 – первоначальный вход в цикл;

2 – вход в цикл при его очередном повторении;

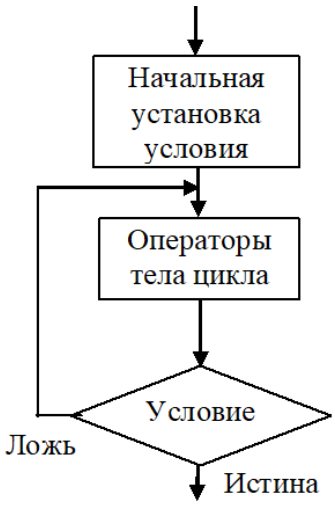
3 – выход из цикла по его окончании.

Если количество повторений цикла определяется в ходе вычислений в зависимости от полученных текущих результатов, используются операторы цикла с предусловием или постусловием (схемы б и в – итерационные циклы).

б) **`while do`**



в) **`repeat until`**



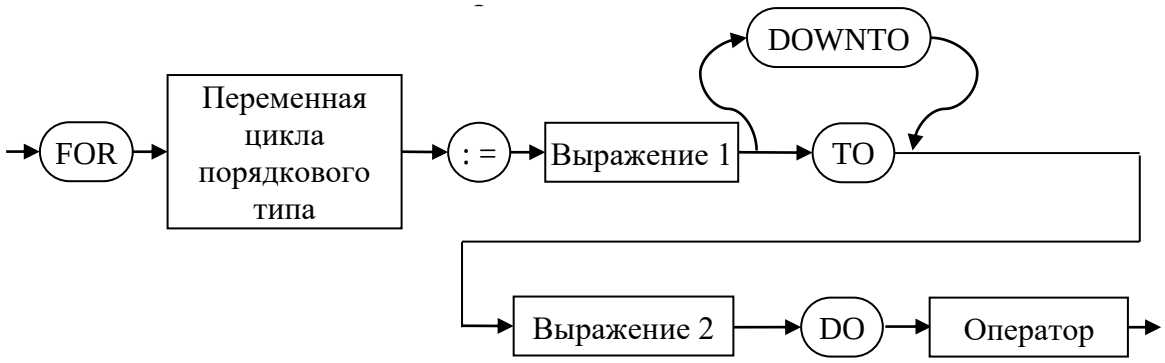
Рассмотрим подробно каждый оператор цикла.

Оператор повторения `for`

Оператор цикла с параметром, или оператор повторения **for**, состоит из заголовка и тела цикла. Он может быть представлен в двух форматах:

1. **`for`** `параметр_цикла:=S1` **`to`** `S2` **`do`** `оператор;`
2. **`for`** `параметр_цикла:=S1` **`downto`** `S2` **`do`** `оператор;`

или синтаксической диаграммой:



Выражения `S1` и `S2` определяют соответственно начальное и конечное значения параметра цикла; **`for`** ... **`do`** – заголовок цикла; `оператор` образует тело цикла, это может быть простой или составной оператор.

Оператор **`for`** обеспечивает выполнение тела цикла до тех пор, пока не будут перебраны все значения параметра цикла – от начального до конечного.

Параметр цикла, его начальное значение `S1` и конечное значение `S2` должны принадлежать к одному и тому же скалярному порядковому типу данных (совместимыми по присваиванию значений).

Количество повторений оператора **`for`** видно из следующей таблицы (таблица 13):

Таблица 13 – Количество повторений оператора for.

Оператор	**S1<S2**	**S1=S2**	**S1>S2**
`for` параметр_цикла:=S1 **`to`** S2 **`do`** оператор;	S2-S1+1 раз	1 раз	не выполняется
`for` параметр_цикла:=S1 **`downto`** S2 **`do`** оператор;	не выполняется	1 раз	S1-S2+1 раз

Замечание 1. Существует *правило*: в теле цикла нельзя самим изменять значение параметра цикла.

Замечание 2. В теле цикла могут быть и другие вложенные операторы цикла **`for`**.

После нормального завершения оператора **`for`** значение параметра цикла обычно равно конечному значению. Если же **`for`** не выполнялся, то значение параметра цикла не определено.

Если нужно преждевременно выйти из цикла, применяется оператор **`break`**, который реализует выход на оператор, расположенный сразу после тела цикла.

Если используются типы `Integer`, `Byte`, интервальный на базе целочисленных типов, значения параметра цикла последовательно

увеличиваются при **for** ... **to** или уменьшаются при **for** ... **downto** на 1 при каждом повторении. Если используются не целочисленные порядковые типы, то увеличение происходит на условную «единицу» определенного типа.

Рассмотрим некоторые примеры использования оператора **for** (таблица 14):

Таблица 14 – Примеры использования оператора for.

Оператор	Результат
`for i:=10 to 14 do Write (i:3);`	`10 11 12 13 14`
`for i:=14 downto 10 do Write (i:3);`	`14 13 12 11 10`
`for ch:='a' to 'e' do Write (ch:2);`	`a b c d e`
`for ch:= 'e' downto 'a' do Write (ch:2);`	`e d c b a`
`var Sort :(S1, S22, S23, S3);` … `for Sort:=S1 to S3 do оператор;`	

Замечание 3. Существует *правило*: в теле цикла нельзя самим изменять значение параметра цикла.

Замечание 4. В теле цикла могут быть и другие вложенные операторы цикла **for**.

После нормального завершения оператора **for** значение параметра цикла обычно равно конечному значению. Если же **for** не выполнялся, то значение параметра цикла не определено.

Если нужно преждевременно выйти из цикла, применяется оператор **break**, который реализует выход на оператор, расположенный сразу после тела цикла.

Типичными задачами использования циклов являются вычисление суммы и возведение в целочисленную степень.

1) Вычисление суммы $S = \sum_{i=1}^{10} \frac{1}{i^2}$.

Решение.

```
… S:=0;
for i:=1 to 10 do S:=S+1.0/Sqr(i);
Write(S:8:4); …
```

2) Возведение в целочисленную степень $y = a^n$.

Решение.

```
… y:=1;
for i:=1 to n do  y:=y*a; …
```

На следующем примере хорошо видно, что не только операторы, но и выбранные типы данных влияют на решение поставленной задачи.

Задача. Вычислить сумму:

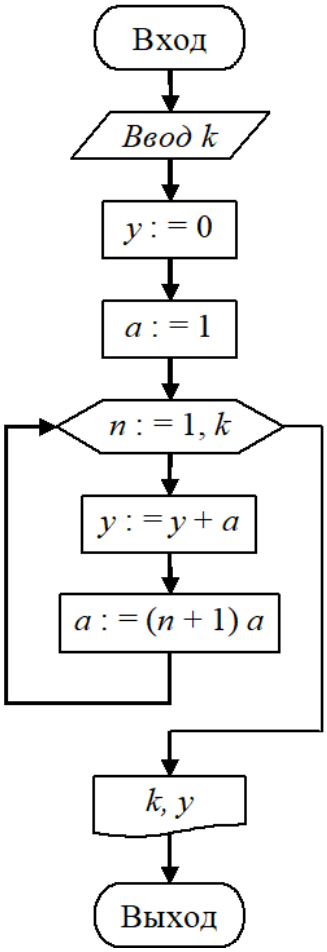
$$y = \sum_{n=1}^{k} n! = 1 + 1 \cdot 2 + 1 \cdot 2 \cdot 3 + 1 \cdot 2 \cdot 3 \cdot 4 + \ldots$$

Иначе:

$$y = \sum_{n=1}^{k} a_n, \qquad a_n = n!, \; a_0 = 1, \quad a_{n+1} = (n+1) \cdot a_n.$$

Поскольку нами ранее рассматривались блок-схемы решения задач по накоплению сумм, сразу получим следующую блок-схему.

Из-за ограниченности диапазонов целочисленных типов возникнет следующая ситуация. Если `a`, `y` имеют тип `Integer`, то для $k = \overline{1,7}$ результат получается правильный, а при $k = 8$ – отрицательный, следовательно, неверный. Если же `a`, `y` имеют тип `Longint`, то при $k = \overline{1,12}$ результат верный, а при $k = 13$ – отрицательный.



Если `a`, `y` имеют тип `Real`, то результат будет приблизительным, ведь для мантиссы отводится ограниченное количество битов.

Можно попробовать тип Comp, однако там также есть ограничения на диапазон данных.

В этом случае для любых больших k нужно использовать массив цифр в 10-й с/с для хранения `a` и реализовать операции сложения и умножения над такими «длинными» числами.

Таким образом, алгоритм может иметь несколько реализаций, в зависимости от типа данных для хранения `a`.

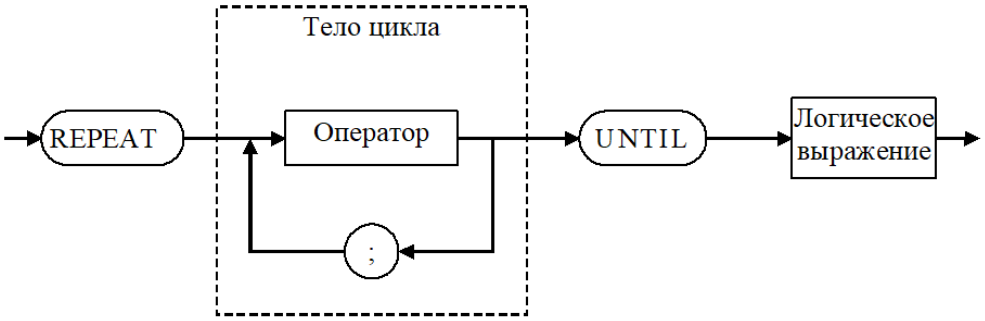
Программа, реализующая алгоритм на языке Pascal:

```
program Sum;
const
    k = 12;
var
    y, a : Longint;
       i : Byte;
begin
 y := 0;
 a := 1;
 for i := 1 to k do
    begin
```

```
      y := y + a;
      a := a * (i+1)
     end;
Writeln ('k=', k, ' y=', y)
end.
```

Оператор повторения repeat

Оператор повторения **repeat** состоит из заголовка цикла (**repeat**), тела цикла и условия завершения (**until**).



Сначала выполняется тело цикла, затем проверяется условие выхода из цикла.

После получения **false** тело активизируется еще раз; после получения **true** происходит выход из цикла. Этот оператор выполняется не менее одного раза! Операторные скобки здесь отсутствуют. По меньшей мере, один из операторов тела цикла должен влиять на значение условия, иначе цикл будет выполняться бесконечно или только один раз.

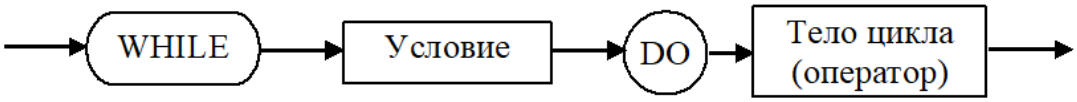
Оператор вида **repeat … until false** выполняется бесконечное число раз, если внутри тела цикла нет операторов **break** или **goto**.

Задача. Подсчитать в некотором тексте, заканчивающимся символом «.» (точка), количество символов ’!’ и ’?’.

```
program Letters;
var ch : Char;
    s  : Integer;
begin
s:=0;
repeat
      Read (ch);  {посимвольно вводится текст}
      if (ch =’!’) or (ch =’?’)  then s:=s+1;
until ch=’.’;
Writeln (’кол-во символов =’, s:3);
end.
```

Оператор повторения while

Оператор **while** подобен оператору **repeat**, но условие выполнения тела цикла проверяется в начале работы оператора.



Если условие истинно, выполняется тело цикла, и так продолжается до тех пор, пока условие не станет ложным. В этом случае выполнение оператора цикла завершается.

```
while  true  do  Write('бесконечный цикл');
while  true = false  do  Write('пустой цикл');
```

Циклы **repeat** и **while** работают с различными условиями: у цикла **repeat** – условие окончания цикла, а у цикла **while** – условие продолжения цикла.

Условие – это логическое выражение, подсчитывающееся при каждом выполнении тела цикла, поэтому для экономии времени подсчет его надо делать максимально простым. Чтобы выйти из цикла **while** преждевременно, нужно условие сделать ложным или использовать оператор **break**.

Задача. Задано действительное a и натуральное k. Подсчитать $b = a^k$ не используя операцию возведения в степень.

Рассмотрим обычные алгоритмы (их как минимум два):

а)
```
const k=...; a=...;
   ...
b:=1; for i:=1 to k do b:=b*a;
```

б)
```
const k:Byte=...; a=...;
   ...
b:=1;
while k>0 do
      begin b:=b*a; Dec(k) end;
```

Однако существует улучшенный алгоритм, который называется *быстрой степенью*.

Быстрая степень

Выше мы рассматривали простой алгоритм:

$$b := 1; \quad \underbrace{k := k - 1; \quad b := b * a;}_{\textit{повторять пока } k>0}$$

В конце работы алгоритма получится $b = a^k$.

Однако, если k – четное, тогда имеет место тождество $a^k = (a^2)^{k/2}$, значит, можно сделать замену $k := k/2$, $a := a * a$ и снова применить данный алгоритм. Получим следующую программу:

```
program Quik_power;
var
     x, a, b : Real;
     k, n    : Integer;
```

```
begin
  Writeln('a, k -'); Readln(x, n);
  Write(x, '^', n, '=');
  k := n;
  b := 1;
  a := x;
  while k>0 do
     begin
      if  Odd(k)  then  b := b * a;
      k := k div 2;
      a := a * a;
     end;
Writeln (b);
end.
```

Здесь использовали сокращенную форму оператора `if`, потому что после нечетного k, равного $k-1$, будет идти четное k и мы выходим на `k := k div 2`, а это целочисленное деление можно делать явно не уменьшая значение k.

Проверим правильность работы программы при $k=7$. $a^7 - ?$

Результат получим за три прохода ($7 < 8 = 2^3$):

$k := 7; \quad b := 1; \quad a := x; \quad \{ \text{if } k - \text{нечетное} \xrightarrow{да} b = a;$

$k = \left[7/2 \right] = 3; \quad a = x^2; \quad \text{if } k - \text{нечетное} \xrightarrow{да} b = x \cdot x^2;$

$k = \left[3/2 \right] = 1; \quad a = x^4; \quad \text{if } k - \text{нечетное} \xrightarrow{да} b = x \cdot x^2 \cdot x^4 = x^7;$

$k = 0 \}; \quad \text{конец}$

Итерационные алгоритмы высшей математики

Как уже отмечалось, при помощи оператора **`for` … `do`** программируются циклы с заданным числом повторений, а операторы **`repeat` ... `until`** или **`while` ... `do`** используются тогда, когда количество повторений наперед неизвестно, а задано некоторое условие его окончания (или продолжения).

В математике существует множество задач, которые можно решать при помощи таких циклов.

Пример 1. Во время повторений тела цикла образуется последовательность значений $y_1, y_2, y_3, \ldots$, стремящаяся к лимиту a: $\lim\limits_{n \to \infty} y_n \to a$.

Каждое новое y_n у такой последовательности определяется с учетом предыдущего y_{n-1} (или предыдущих k членов: $y_{n-1}, y_{n-2}, \ldots, y_{n-k}$), и является по сравнению с ними более точным приближением к искомому результату (лимиту) a.

Циклы, реализующие такую последовательность приближений (итераций), называют *итерационными*.

В итерационных циклах условие продолжения (окончания) цикла основывается на свойстве безграничного приближения значений y_n к пределу a при увеличении n.

Итерационный цикл заканчивают, если для некоторого значения n выполняется условие

$$|y_n - y_{n-1}| \le \varepsilon,$$

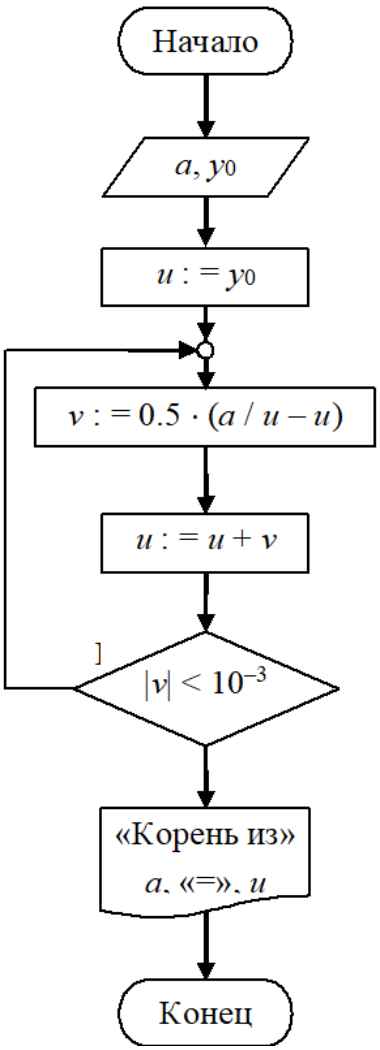
где ε – допустимая погрешность подсчета результата.

Тогда результат отождествляют со значением y_n, т.е. считают, что $y_n \approx a$.

Замечание. Последовательность $\{y_n\}$ не нужно путать с массивом. В массиве количество элементов известно заранее, а в последовательности количество элементов фактически неограниченно, да и элементы последовательности получаются рекуррентно, поэтому предыдущие элементы последовательности можно «долго» не хранить в памяти.

Пример 2. Из высшей математики известно, что для нахождения квадратного корня $y = \sqrt{a}$, $a \ge 0$, можно применить следующий алгоритм: найти с точностью до $\varepsilon > 0$ предел последовательности $\{y_n\}$, где при заданном y_0

$$y_{n+1} = y_n + \frac{1}{2}\left(\frac{a}{y_n} - y_n\right), \quad n = 0,1,2,...$$



Исходя из предыдущих рассуждений, напишем программу нахождения $\sqrt{a}$ с точностью $\varepsilon = 10^{-3}$.

Решение. Исходными данными являются: число a>0, из которого нужно извлечь корень; значение y_0>0, которое является некоторым грубым приближением к корню (его можно подсчитать, например, графически).

Результатом будем считать тот y_{n+1}, при котором $|y_{n+1} - y_n| \equiv \left|\frac{1}{2}\left(\frac{a}{y_n} - y_n\right)\right| < \varepsilon$.

Для хранения элементов последовательности $\{y_n\}$ заведем одну переменную u.

Программа.

```
program Newton;
var
   a, y0, u, v : Real;
   n           : Integer;
```

```
begin
  Writeln('a, y0 =');
  Readln(a, y0);
  u := y0;
  n := 0;

  repeat
      v := 0.5 * (a / u - u);
      Inc(n);
      u := u + v;
  until (Abs(v) < 1E-3) or (n > 100);
  Writeln('корень з', a : 8 : 3, '=', u);
end.
```

Замечание. В таких задачах, когда плохо подобрано начальное приближение y_0, может произойти «зацикливание», поэтому нужно предусмотреть подсчет шагов, и когда их стало много (например, n>100), тогда процесс прерываем, печатаем v, u, n, а потом оцениваем, правильно ли был произведен выбор начального приближения.

Типичным примером итерационного циклического процесса может служить задача вычисления суммы бесконечного ряда. Понятие суммы ряда связано с понятием сходимости бесконечного ряда.

Ряд – это бесконечная сумма вида

$$a_1 + a_2 + a_3 + ... + a_n + ... = \sum_{k=1}^{\infty} a_k \qquad (4)$$

Слагаемые a_k называются членами ряда, а суммы конечного числа членов ряда $S_n = a_1 + ... + a_n, \quad n = 1,2,...$ – частичными суммами ряда порядка *n*.

Различают числовые и функциональные ряды.

Примеры числового ряда:

$$\frac{\pi}{3} = 1 + \frac{1}{2^3 \cdot 3} + \frac{3}{2^7 \cdot 5} + \frac{5}{2^{10} \cdot 7} + ...,$$

сумма геометрической прогрессии $1 + q + q^2 + ... + q^n + ... = S$,

$S = \frac{1}{1-q}$, когда $|q| < 1$.

Членами функционального ряда являются функции.

Пример функционального ряда:

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

Ряд (4) называется сходящимся, если сходится последовательность его частичных сумм $\{S_n\}$. В этом случае предел $\lim\limits_{n\to\infty} S_n = S$ называется суммой ряда. Если лимит не существует, тогда ряд расходящийся.

Существуют оценки сходимости рядов. Одну из них используют при численных подсчетах суммы ряда.

Признак Лейбница: если $\lim\limits_{n\to\infty} a_n = 0$, $a_n \geq a_{n+1} > 0$, тогда знакочередующийся ряд $\sum\limits_{n=1}^{\infty} (-1)^n a_n$ – сходится.

В этой связи ряд заменяется его конечной суммой S_n, где n – это номер того слагаемого, для которого $|a_n| < \varepsilon$.

Все тригонометрические функции, а также логарифмическая и экспоненциальная функции в языках программирования подсчитываются при помощи разложения их в ряд.

Рассмотрим подробности подсчета частичных сумм вида: $S_N(x) = \sum\limits_{n=1}^{N} a_n(x)$.

Обычно формула общего члена суммы $a_n(x)$ принадлежит к одному из следующих трех типов:

а) $\frac{x^n}{n!}$; $(-1)^n \frac{x^{2n+1}}{(2n+1)!}$; $\frac{x^{2n}}{(2n)!}$ – *получение произведения*;

б) $\frac{\cos nx}{n}$; $\frac{\sin(2n-1)x}{2n-1}$; $\frac{\cos 2nx}{4n^2-1}$ – *произведение отсутствует совсем*;

в) $\frac{x^{4n+1}}{4n+1}$; $(-1)^n \frac{\cos nx}{n^2+1}$; $\frac{n^2+1}{n!}\left(\frac{x}{2}\right)^n$ – *произведение присутствует частично*.

В случае а) для подсчета слагаемых суммы целесообразно использовать рекуррентное соотношение, т.е. $a_{n+1}(x)$ выражать через $a_n(x)$ как $a_{n+1}(x) = \varphi_n \cdot a_n(x)$, φ_n – определяется из поставленной задачи. Это позволяет избежать пересчета величин, например, $x^{n+1} = x \cdot \left(x^n\right)$ (так как x^n было известно). Кроме того, подсчет члена суммы по общей формуле в ряде случаев совершенно невозможен. Например, если присутствует $n!$, то нужно помнить, что подсчет по определению $n!$ быстро приводит к переполнению.

В случае б) каждый член суммы целесообразно подсчитывать по общей формуле.

В случае в) член суммы нужно представить как произведение двух сомножителей: один вида а), второй – б), а затем раздельно их подсчитывать.

Например:

$$\frac{n^2+1}{n!}\left(\frac{x}{2}\right)^n = \left(n^2+1\right)\left[\frac{\left(x/2\right)^n}{n!}\right].$$

Здесь $\left(n^2+1\right)$ имеет вид б), а $\left[\frac{\left(x/2\right)^n}{n!}\right]$ – вид а).

Задача. Для зафиксированного x подсчитать с погрешностью $\varepsilon = 10^{-4}$ значение функции $S(x) = \cos x$, используя известное (из справочников) разложение функции $\cos x$ в следующий ряд:

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ... = \sum_{n=0}^{\infty} a_n(x),$$

где $a_n(x) = (-1)^n \frac{x^{2n}}{(2n)!}$, $0! = 1$.

Решение. Будем подсчитывать $S_N(x) = \sum_{n=0}^{N} a_n(x)$ до тех пор, пока еще $|S_{N+1} - S_N| > \varepsilon$, или $|a_{N+1}(x)| > \varepsilon$.

Предыдущая классификация типов слагаемых подсказывает, что общий член нашего ряда принадлежит к типу а). Значит, будем искать зависимость между соседними слагаемыми.

Пусть зафиксировано некое n. Тогда $a_0 = 1$, $a_{n+1}(x) = \varphi_n \cdot a_n(x)$,

где $a_n(x) = (-1)^n \frac{x^{2n}}{(2n)!}$. И можем получить

$$\varphi_n = \frac{a_{n+1}(x)}{a_n(x)} = \frac{(-1)^{n+1} \cdot x^{2n+2}}{(2n+2)!} \cdot \frac{(2n)!}{x^{2n} \cdot (-1)^n} = \frac{-x^2}{(2n+1)(2n+2)}.$$

Фрагмент программы, реализующей алгоритм на языке Pascal.

```
{Здесь объявление переменных и начало программы}
S := 0;
a := 1;
k := 1;                                          {n := 0;}
y := -x * x;
repeat
   S := S +a;
   a := a * y / k / ( k+ 1);
     {a := -a * Sqr(x) / ((2 * n + 1) * (2 * n + 2));}
   k := k + 2;                                   {n := n + 1;}
```

```
until abs(a) < eps;
{напечатать результат}
```

Для оптимизации цикла при записи программы сразу улучшили вычислительную схему алгоритма.

1.10. Структуры данных и работа с ними

Содержание

- Порядковые типы.
- Множества.
- Массивы.
- Строки символов.
- Записи.

Порядковые типы

В группу порядковых типов данных объединены целочисленные, символьный, логический, перечислимый и интервальные типы, потому что они обладают следующими свойствами:

1) все возможные значения данных порядкового типа представляют собой ограниченное упорядоченное множество;
2) к данным любого порядкового типа можно применять стандартную функцию `Ord`, которая в качестве результата возвращает порядковый номер конкретного значения в данном типе;
3) к данным любого порядкового типа можно применять стандартные функции `Pred` и `Succ`, которые возвращают соответственно предыдущее и последующее значения;
4) к данным любого порядкового типа могут быть применены стандартные функции `Low` и `High`, которые возвращают наименьшее и наибольшее значения величин данного типа.

Структурированные типы данных определяют упорядоченную совокупность скалярных данных и характеризуются типом своих компонент.

Множества

Тип «множество» образуется специальной конструкцией:

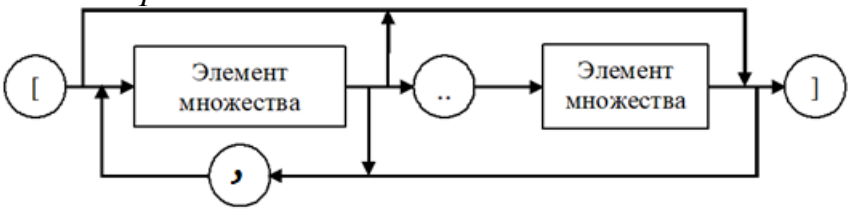
```
set of базовый_порядковый_тип
```

Множество в языке Pascal – это ограниченная (<= 256 элементов) совокупность попарно различных объектов одного скалярного порядкового типа, который называется базовым. В качестве базового может быть любой порядковый тип с элементами, для которых функция `Ord` возвращает значения в диапазоне от 0 до 255. Скалярные типы `Byte` и `Char` могут быть основой для построения множеств.

Базовый тип задается перечислением, диапазоном или именем.

Каждый объект в множестве называется *элементом множества*. Если множество не имеет элементов, оно называется пустым. Область значений типа «множество» – набор всевозможных подмножеств, которые составлены из элементов базового типа.

Изображение константы-множества:



Элемент множества – константа, переменная, выражение.

Попытка объявленным переменным присвоить значения, которые не входят в базовое множество, не вызывает программное прерывание, просто такие значения игнорируются. Например:

```
type
    SetOfChar  = set of Char; {множество из символов }
    SetOfByte  = set of Byte; {множество из чисел  }
    SetOfDigit = set of 0..9; {множество-интервал чисел }
    SetOfDChar = set of '0'..'9';
                            {множество-интервал символов}
var
         SChar  : SetOfChar;
         SByte  : SetOfByte;
         SDigit : SetOfDigit;
    begin
           SDigit := [0,2,4,6,8];
           SDigit := [10];   {ошибка не фатальная}
           SChar  := ['a'..'z','A'..'Z'];
           SChar  := ['2'];  {ошибка не фатальная}
           SChar  := [2];
                     {type mismatch – не совпадение типов}
          …
```

Порядок чередования элементов при их перечислении не имеет значения, также не имеет значения количество повторений элемента.

Нет оператора для извлечения элемента множества, но можно проверить входит ли элемент в множество.

Например:

```
SChar := ['a', 'a', 'a', 'a']
```

эквивалентно однократному упоминания символа `'a'`.

Например, после объявления переменных

```
var
     x : Byte;
     S : set of Byte;
```

возможны следующие присваивания:

```
     x := 3;
     S := [1, 2, x];
     S := S+[x+1];
```

Множество может расширяться или сокращаться по ходу программы. Процедура `Include (S, i)` добавляет в множество `S` элемент, принадлежащий базовому типу множества, заданный параметром `i`. Процедура `Exclude (S, i)` удаляет из множества `S` элемент, принадлежащий базовому типу множества, заданный параметром `i`.

Операции над множествами, которые определены в языке Pascal, представлены в следующих трех таблицах: (таблицы 15-17):

Таблица 15 – Объединение множеств.

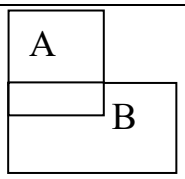
Объединение множеств (то, что входит в 1-е и 2-е множества)				Рисунок
В математике		В языке Pascal		
Обозначение	Пример	Обозначение	Пример	
$\cup$	$A \cup B$	+	$A+B$	

Таблица 16 – Пересечение множеств.

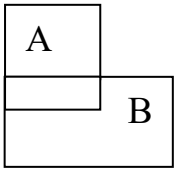
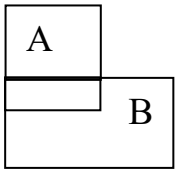
Пересечение множеств (то, что общее в 1-м и 2-м множествах)				Рисунок
В математике		В языке Pascal		
Обозначение	Пример	Обозначение	Пример	
$\cap$	$A \cap B$	*	$A*B$	

Таблица 17 – Разность множеств.

Разность множеств (то, что есть только в 1-м множестве, за исключением того, что есть в обоих)				Рисунок
В математике		В языке Pascal		
Обозначение	Пример	Обозначение	Пример	
\	$A \setminus B$	–	$A-B$	

Результат этих операций – всегда множество.

Вторая группа операций – это операции сравнения или отношений: =, <>,> =, <=. Результат всегда **`true`** или **`false`** (таблица 18):

Таблица 18 – Операции сравнения.

Название		Форма	Пояснения: `false` – в противном случае
=	Проверка на равенство	`S1=S2`	**`true`**, когда `S1` и `S2` состоят из одинаковых элементов
<>	Проверка на неравенство	`S1<>S2`	**`true`**, когда `S1` и `S2` отличаются хоть одним элементом
<=	Проверка на подмножество ⊂	`S1<=S2`	**`true`**, когда все элементы `S1` входят в `S2`, независимо от порядка чередования
>=	Проверка на подмножество ⊃	`S1>=S2`	**`true`**, когда все элементы `S2` в `S1`
in	Проверка на вхождение элемента в множество	`E in S1` `E in […]`	**`true`**, когда значение элемента `E` принадлежит базовому типу множества и входит в множество `S1` или в множество-константу ([...])

Операцию **`in`** хорошо использовать, когда нужно проверить попадание значения в диапазоны перечислимых типов:

```
if ch in ['a'.. 'x', 'A'.. 'X'] then …;
if j in [100..200] then …;
```

Здесь использованы константы типа «множество». Этим упрощаются логические операторы. Сравните следующее условие:

```
C in ['0'.. '9', 'A'.. 'Z']
```

и такое

```
(C>='0') and (C<='9') or (C>='A') and (C<='Z')
```

Достоинства множеств очевидны:

1) экономится память, так как объем памяти, который занимает один элемент множества, равен 1 биту. Значит, объем множества из 256 элементов составляет всего 32 байта;
2) экономится время, так как разработчики трансляторов обычно связывают элементы множества с машинным словом, в котором: код «1» – присутствие элемента во множестве, «0» – отсутствие.

Типизированные константы типа «множество»

Пример.

```
const   Xset : set of Char=['a', '6', 'в', 'A'..'B'];
```

Рассмотрим пример использования данных типа множество.

Задача 1. Запрограммировать решето Эратосфена для поиска простых чисел, не превосходящих 255.

Алгоритм. Среди всей совокупности чисел от 2 до 255 вычеркнем кратные очередному простому числу.

```
program Simple;
var
     Prime   : set of 2..255;
     j, i, m : Integer;
begin
Write ('введите m<=255');
Readln (m);
if    m<2   then m:=2;
Prime := [2..m];
for i := 2 to m do
    if i in Prime then
       for j := 2 to (m div i) do
           Prime := Prime-[i * j];
            {исключили все кратные простому числу i}
Writeln ('простые числа <=', m);
for i := 2 to m do
    if i in Prime then Write (i:4);
end.
```

Задача 2. Запрограммировать решето Эратосфена для поиска простых чисел, не превосходящих 512.

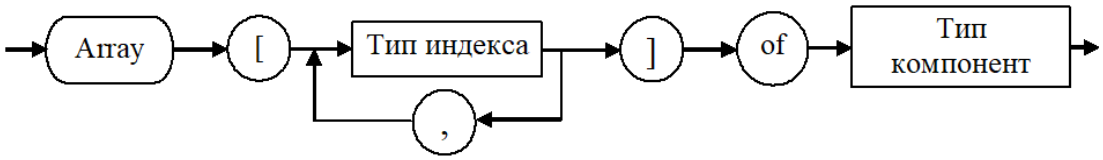
```
program Simple_2Set;
var
     Prime2_255,  Prime256_511 : set of 0..255;
     j, i, m : Integer;
begin
 Prime2_255:=[];
 Prime256_511:=[];
 Writeln ('введите m<512');
 Readln (m);
 if m>512 then m:=512;
 if m<2   then m:=2;
 case m of
     0..255 : Prime2_255   := [2..m ];
   256..511 : begin Prime2_255   := [2..255 ];
                    Prime256_511 := [0..m - 256 ];
              end;
 end;
 for i := 2 to m do
  if i in [0..255]  then
   begin
      if i in Prime2_255 then
```

```
       for j := 2 to (m div i) do
        case i * j  of
          0..255 : Prime2_255   := Prime2_255-[i * j];
        256..511 : Prime256_511:=Prime256_511-[(i*j)-256];
        end;
   end;
 Writeln ('Prime chisla <=', m);
 for i := 2 to m do
  case i of
   0..255 : if i in Prime2_255   then Write(i:8);
   256..511:if (i-256) in Prime256_511 then Write(i:8);
  end;
end.
```

Массивы

Переменные типа массив или *массив* это упорядоченная совокупность однотипных данных, которые хранятся последовательно.

Тип массив определяется синтаксической диаграммой



Тип индекса – порядковый (кроме `Longint`), он показывает, в каких границах изменяется индекс массива. Тип компонент – любой известный тип данных.

Массив обязательно имеет фиксированные размеры, которые становятся известными по типу индексов, и которые определяют, сколько элементов хранится в нем. Любой элемент в массиве можно найти по его индексу.

Индексы – это выражения указанного порядкового типа.

Если в описании массива задан один индекс, массив называется *одномерным*, два – *двумерным*, `n` – *n-мерным*. Размеры массива ограничены только объемом памяти.

В секции **`type`** можно определить типы массивов. Это именованные типы массивов, которые затем используются при определениях переменных типа «массив».

После объявления массива каждый его элемент можно обрабатывать, указав имя (идентификатор) массива и индексы элемента в квадратных скобках.

Индексированные элементы массива называются *индексированными переменными* и могут быть использованы как простые переменные.

```
type
 Array01to10 = array [ 1..10] of Real;
 Array11to20 = array [11..20] of Real;
```

Эти два типа `Array01to10` и `Array11to20` по-разному нумеруют свои элементы, хотя оба типа содержат наборы из 10 значений типа `Real`.

```
var
    a01to10 : array01to10;
    a11to20 : array11to20;
```

Доступ к *i*-му элементу массива:

`a01to10[i], i=1,…,10,`

`a11to20[i], i=11,…,20.`

Пример 1.

```
const   k=4; n=6;
type    TVector  = array [1..4] of Integer;
        TMatrica = array [-4..4, -4..4] of Real;
        TMassiv = array [1..4] of TVector;
var     Matr : TMatrica;
        Vect : TVector;
        M : TMassiv;
        Mas : array [1..k, 1..n] of Char;
```

Доступ к элементу mas_{ij} → `mas[i,j]`. Доступ к элементам массива `M` можно осуществить и так: `m[i][j]`, и так: `m[i,j]`.

Пример 2.

```
type
   MonthType    = (January, February, March, April, May);
   ComplectType = array [MonthType] of Word;
   SpringType   = array [March .. May] of Word;
var
    Complect  : ComplectType;     {5 элементов типа Word}
    Spring    : SpringType;       {3 элемента  типа Word}
    Alpfa     : array ['A' .. 'Z'] of Char;   {26 элементов}
    Switch    : array [Boolean] of Byte;    {2 элемента  }
```

Обращение к элементам описанных массивов:

```
Spring[April];  Alpfa['X']; Switch[true];
```

Пример 3.

```
var
   M : array[-10 .. 0, 'A' .. 'C', Boolean] of Byte;
```

Эквивалентная запись:

```
M : array[-10..0] of array['A'..'C'] of
                     array[Boolean] of Byte;
```

В последнем случае `M[0]` – массив-матрица типа

```
array ['A' .. 'C'] of array [Boolean] of Byte;
```

или

`M[0] –  array ['A' .. 'C', Boolean] of Byte.`

`M[0, 'B']` – это вектор типа `array [Boolean] of Byte`.

M[0, 'B', false] – значение типа Byte.

Использовать элементы M[0], M[0, 'B'] довольно сложно, так как нужно заботиться о совместимости типов данных.

В памяти компьютера массивы хранятся как последовательности компонентов, при этом, если в массиве несколько индексов, тогда в первую очередь изменяется последний индекс, потом предпоследний и так далее до первого. Отсюда следует, что матрицы хранятся по строкам:

```
A : array[1..5, 1..5] of Byte;
{a[1,1], a[1,2], a[1,3],…, a[1,5], a[2,1],…, a[5,5]}
```

Адрес начала массива в памяти соответствует адресу его первого элемента, т.е. элемента с минимальными значениями индексов.

Если программа или фрагмент ее компилируется в режиме {$R+}, тогда при обращении к элементам массивов будет проверяться принадлежность значения индекса объявленному диапазону, и в случае ошибки (нарушение границ диапазона) возникнет ошибка Range check Error.

Например, после объявления

```
v : array [1..10] of Integer;
```

обращение к элементу v[11] – даст ошибку на шаге компиляции в режиме {$R+}.

Если в режиме {$R-} возникнет аналогичная ситуация, программа будет работать дальше, и некорректное значение индекса может получить какое-нибудь значение, но все же не из нужного массива. Обычно программу отлаживают в режиме {$R+}, а эксплуатируют при режиме {$R-}.

Работа с элементом массива занимает больше времени, чем со скалярной переменной, так как для доступа к элементу массива неявно вычисляется местоположение элемента в памяти. Но при использовании массивов программы становятся более функциональными.

Действия над массивами

Переменная типа массив может участвовать только в операторе «присвоить значение» :=.

Массивы, участвующие в этой операции, должны быть идентичными по структуре.

Для массивов, объявленных с использованием служебного слова **array**, идентичность по структуре – это фактически объявление массивов одним списком, что не обязательно при объявлении массивов с использованием пользовательского типа массива.

Например, рассмотрим следующие объявления:

```
type
       mas = array [1..20] of Real;
var
      A, B : array [1..20] of Real;
      X    : mas;
      C, D : array [1..5, 1..5] of Word;
      V    : array [1..20] of Real;
```

```
      Z    : mas;
```

Оператор `A := B` означает, что соответствующим элементам массива `A` присваиваются значения элементов массива `B`. Оператор `C := D` означает, что соответствующим элементам массива `C` присваиваются значения элементов массива `D`. Операторы `X := A` и `V := A` приведут к ошибкам компиляции, т.к. для компилятора это разные массивы. Но присваивание `X := Z` будет корректным, т.к. эти массивы объявлены при помощи идентификатора типа.

Действия над элементами массива

Рассмотрим типичные ситуации, возникающие при работе с данными типа «массив» в следующей программе:

```
const
      n=10; m=5; L=4;
var
          A, O : array [1..L] of Real;
             B : array [1..n,1..m] of Integer;
       k, i, j : Integer;
             S : Real;
            . . .
               {инициализация одномерного массива}
for i:=1 to L do Read(A[i]);

{инициализация двумерного массива }
for i:=1 to n do
    for j:=1 to m do Read(B[i,j]);

{вывод двумерного массива в виде матрицы}
for i:=1 to n do
  begin
    for j:=1 to m do Write(B[i,j]);
    Writeln;
  end;
```

Переменные типа «массив» со стартовым значением

Стартовые значения сложных переменных задаются по-разному для разных типов. Массивы задаются перечислением их элементов в круглых скобках. Если массив многомерный (массив массивов), тогда перечисляются элементы массива, состоящие из элементов-скаляров. Это выглядит таким образом:

```
type
          dim10    = array [1..10]            of Real;
          dim3x6   = array [1..3, 1..6]       of Real;
          dim4x3x2 = array [1..4, 1..3, 1..2] of Word;
const
      D10    : dim10=
             (0, 2.1, 3, 4.5, 6, 7.70, 8, 9.0, 10, 34);
```

```
                        {это один набор из 10 чисел}
D3x6   : dim3x6=
             ((1, 1, 1, 1, 1, 1),
              (2, 2, 2, 2, 2, 2),
              (3, 3, 3, 3, 3, 3));
                         {это три набора по 6 чисел}
D4x3x2 : dim4x3x2=
            (((1,2), (11,22), (111,222)),
             ((3,4), (33,44), (333,444)),
             ((5,6), (55,66), (555,666)),
             ((7,8), (77,88), (777,888)));
        {это четыре набора по три массива из 2 чисел}
```

Замечание. При задании структур типа `array of Char`, базирующихся на символах, можно не перечислять символы, как в следующем примере

```
const CharArray : array[1..5] of Char=('a','b','c','d','e');
```

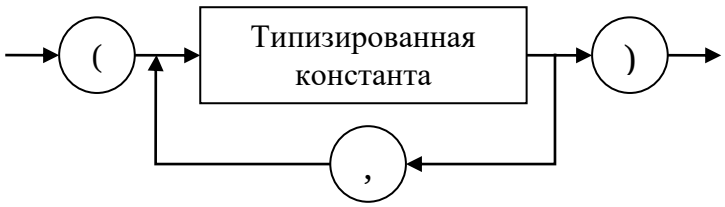
а соединить их в одну строку нужной длины:

```
const  CharArray : array [1..5] of Char='abcde';
```

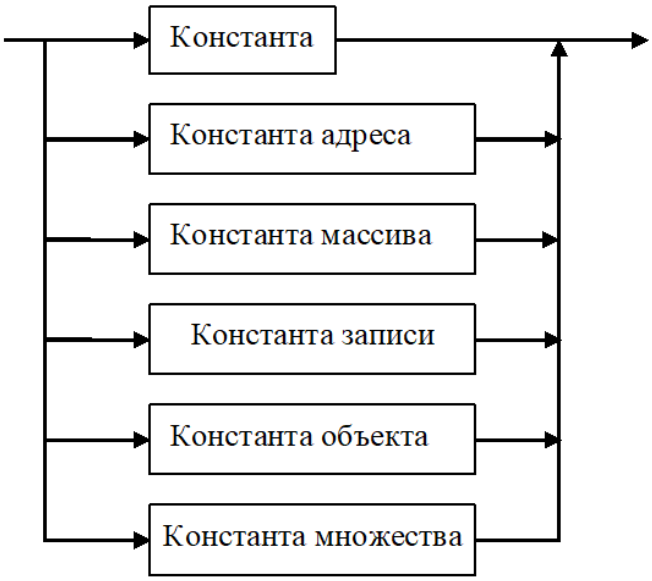
Константы типа массив

Чтобы определить общий случай описания типизированных констант, рассмотрим структурные диаграммы.

Константа-массив:



Типизированная константа:



Задача. Известно, что элементы целочисленного массива A[1..n] не меньше -2 и не больше 12. Подсчитайте, сколько раз в массиве A встречается каждый из элементов.

Решение: Удачно объявив массивы, можно обойтись без операторов ветвления.

```
program Pr;
const
        n=10;
type
        index=-2..12;
const
        A : array [1..n] of index=
                        (0, -2, 5, 1, 0, -2, 5, 8, 1, 0);
var
        B : array [index] of Byte;
        I : index;
        j : 1..n;
begin
  for i:=-2 to 12 do
    B[i]:=0;
  for j:= 1 to n  do
    Inc(B[A[j]]);
       {печать}
  Writeln('элемент':10,'количество':20);
  for i:=-2 to 12 do
    if B[i]<>0 then Writeln(i:10, B[i]:20);
end.
```

Задание. Не различая строчные и прописные буквы латинского алфавита, подсчитать в некотором тексте количество вхождений каждой буквы.

Строки символов

Объявление строковых переменных:

```
var идентификатор : String [max_длина];
```

или

```
var идентификатор : String;
```

Ограничения: 0≤ max_длина ≤255. Когда при определении опущена длина строки, тогда по умолчанию она составляет 255. Так как длина может быть разной, значит, String – это динамические строки.

Приведем различные примеры представления строк.

1. *Первая форма записи:*

```
'Номер п/п | Ф.И.О. | Должность      |'
```

где | – символ вертикальной черты имеет код #179 в альтернативной кодовой таблице.

2. *Вторая форма записи:*

```
'Номер п/п '#179' Ф.И.О. '#179' Должность     '#179
```

3. Строка `#7#32#179#32#32#179` эквивалентна строке `^G' |  |'`

Тип `String` без длины является базовым строковым типом, и он совместим со всеми производными строковыми типами.

При попытке записать в переменную строку больше, чем объявлено в определении строки, лишняя справа часть отсекается.

Когда в переменную пишется строка, короче объявленной, запоминается ее текущая длина.

Строки различных длин совместимы между собой в операторах присвоения значения и в операциях сравнения.

Присваивание значения строкам

Примеры присваивания значения строковым переменным:

```
var  A : String[2]; …
     A :='группа 1';    {⇒  A:='гр' (и длина 2)}
var  A : String[6]; …
     A :='группа 1';     {⇒ A:='группа' и длина строки 6}
var  A : String[10]; …
     A :='группа 1';    {⇒  A:='группа 1' (и длина 8)}
```

Любой символ в строке можно получить по его номеру, например `A[5]`.

Каждая строка всегда «знает», сколько символов в ней содержится в данный момент, потому что символ `S[0]` содержит код символа, который равен числу символов в значении строки `S`. Это значит, что длина строки `S` всегда равно `Ord(S[0])`. Есть встроенная функция `length(S)`, которая возвращает текущую длину строки.

Вывод: Для определенной строковой переменной `S` длиной `N` символов отводится `N+1` байт памяти, из которых нулевой байт содержит длину строки, а остальные `N` байт отведены для символов строки.

Строковые выражения

Рассмотрим операции, процедуры и функции, используемые для работы со строками.

1. Операция сцепления (соединения) строк: +.

Вместо операции + можно использовать встроенную функцию `Concat(str1, str2, …, strn)`, выполняющую сцепление строк `str1, ..., strn`.

2. Операции отношений: >, =, <>, <, <=, >=.

Сравнение строк происходит посимвольно, начиная с первого символа в строке. Строки считаются равными, если выполняются два условия: 1) строки имеют одинаковую длину и 2) содержимое строк посимвольно равно (сравниваются коды). Если при посимвольном сравнении обнаружится, что

один символ больше другого (его код больше), тогда строка, его содержащая, также считается большей.

Примеры.

```
'abcd' = 'abcd'    ⇒  true
'abcd' <> 'abcde'  ⇒  true
'abcd' > 'abcD'    ⇒  true            ('d'>'D')
'abcd' > 'abc'     ⇒  true
'3' > '234'        ⇒  true
```

Редактирование строк

1. Функция `Copy(S, Start, N)` – выделяет из строки `S` подстроку длиной `N` символов, начиная с позиции `Start` (параметры `Start`, `N` – типа `Integer`).

Если значение `Start` превосходит длину строки `S`, тогда результатом будет пустая строка. Если значение `N` большее, чем количество символов от `Start` до конца строки `S`, тогда вернется остаток строки `S` от позиции `Start` до конца строки.

Пример использование функции Copy:

```
SCopy := Copy('ABC***123', 4, 3);          {SCopy='***'}
SCopy := Copy('ABC', 4, 3);                {SCopy=''}
SCopy := Copy('ABC***123', 4, 11);         {SCopy='***123'}
```

2. Функция `Pos(Subs, S)` возвращает номер символа в строке `S`, начиная с которого в строку `S` входит подстрока `Subs` (тип результата `Pos` – `Byte`). Если `S` не содержит `Subs`, тогда функция вернет `0`.

Пример использования функции Pos:

Пусть: `S:='abcdef'`.

```
Pos('de', S) ⇒ 4.
Pos('r', S)  ⇒ 0.
```

3. Процедура `Delete(S, Start, N)` удаляет `N` символов строки `S`, начиная с позиции `Start`. Когда `Start=0`, или превышает длину строки `S`, тогда строка не изменится; если `N` = 0, тогда строка не изменится; если `N` превосходит остаток строки, будет удалена подстрока от `Start` и до конца строки `S`.

Пример использования процедуры Delete:

```
S := 'строка';
Delete(S, 2, 3);       {в S будет строка 'ска'}
S := 'Пример использования процедуры';
Delete(S, 7, 255);  {укорачивается строка S до 6 символов}
                       {в S будет строка 'Пример'}
```

4. Процедура `Insert(Subs, S, Start)` вставляет подстроку `Subs` в строку `S`, начиная с позиции `Start`. Если строка-результат после вставки имеет длину, превышающую длину строки `S`, то строка-результат автоматически укорачивается до объявленной длины `S` (отсекается правый конец).

Пример использования процедуры Insert:

```
S:='Начало-конец';
Insert('середина-', S, 8);
```

Получим результат:

`S='Начало-середина-конец'.`

5. Процедура заполнения `FillChar(V, Len, C)`, где `V` – переменная любого типа, `Len` (тип `Word`) – число байт переменной `V`, которые будут заполнены значением `C` (`C` – типа `Byte` или `Char`).

Пример использования процедуры FillChar:

```
var S: String;
      . . .
     FillChar(S[1], 80, ' ');
     S[0]:=chr(80);
```

Замечание. В случае использования процедуры `FillChar` обязательно нужно в нулевой символ строки записать длину полученной строки.

Преобразование строк

1. Процедура `Str(V, S)` преобразует числовое значение V в строку S. После параметра V может быть указан формат в виде `V:m` или `V:m:n`, где `n < m`, `m`, `n` – данные целого типа; `m` – ширина поля для числа; для вещественного числа `n` – количество знаков после десятичной точки.

Для целых чисел задают только поле `m`. Если же для действительных чисел задают только `m`, тогда число представится в экспоненциальной форме.

Использование функции Str:

```
Str(6.66:8:2, S);          ⇒          {S='    6.66'}
Str(6.66:8:0, S);          ⇒          {S='       7'}
Str(6.66:-8:2, S);         ⇒          {S='6.66'}.
```

Форматы можно задавать и так:

```
var    F, n : Integer;
       S    : String;
         . . .
       F:=-5;    n:=1;
       Str(-123.426:F:n,S);              {S='-123.5'}
```

2. Процедура `Val(S, V, ErrorCode)` преобразует значение строки `S` в величину целочисленного или действительного типа и записывает в числовую переменную `V`. Переменная `ErrorCode` типа `Integer`. Если преобразование возможно, тогда переменная `ErrorCode` равна `0`, в противном случае переменная `ErrorCode` имеет номер позиции, в которой произошла ошибка при преобразовании.

Использование функции Val:

```
var V:Real;  S:String; C:Integer;
S:='014.2E+2';    Val(S,V,C);    {V=1.4200000000E+03; C=0}
S:='014.2A+2';    Val(S,V,C);    {C=6}
```

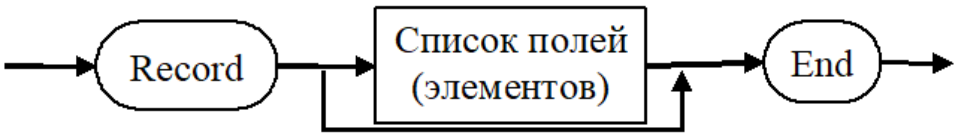
Комбинированный тип «запись»

Кроме рассмотренных выше типов данных существует *комбинированный тип* данных, где объединяются разно-типовые элементы.

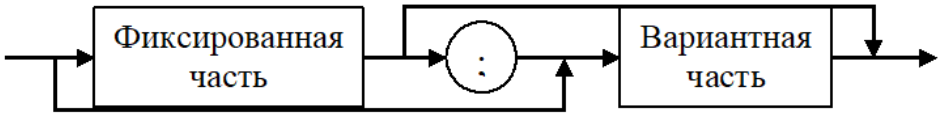
Запись – это объединение данных разных типов, посредством задания именованных полей. В отличие от массивов и множеств поля записей могут иметь различные типы.

Для определения записи применяют служебные слова **record** и **end**.

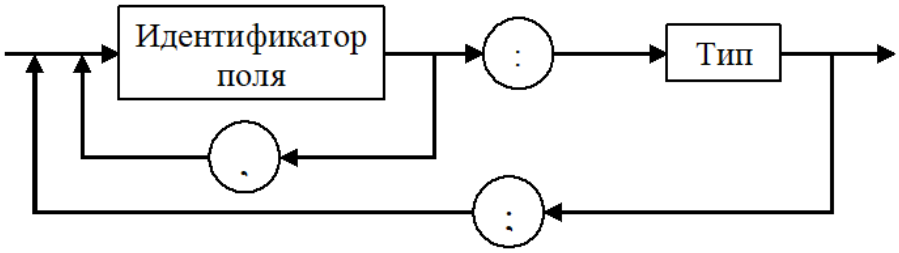
Запись:



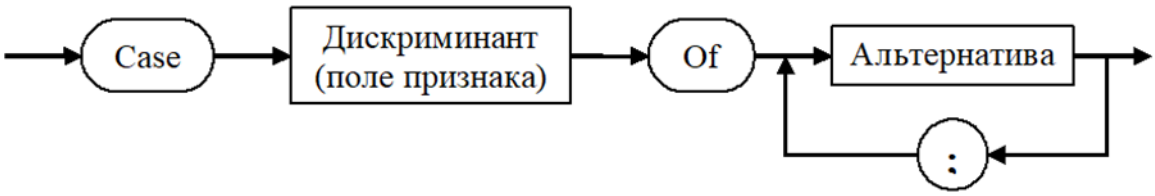
Список полей:



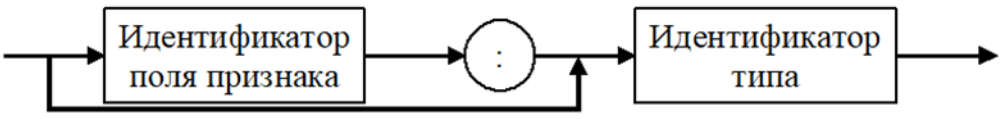
Фиксированная часть:



Вариантная часть:



Дискриминант:



Пример 1.

```
type
     Zap = record
          x, y : Real
         end;
var A : Zap;
```

Пример 2.

```
type
     Person=record
            F_I_O     : String;
                  {фамилия, имя, отчество ≤ 255 символов}
            Ves, Rost : Real;
            Telephone : Longint;    {номер телефона}
            Ozenki    : array[1..4] of Byte;
     end;
```

Тип Person сейчас задает анкету из строки (F_I_O), двух чисел с дробной частью (Ves, Rost), одного целого длинного числа (Telephone) и массива на четыре байта (Ozenki).

Если задано объявление

```
var  Kto_to : person;
```

то под именем Kto_**to** понимается данное с конкретными значениями.

Доступ к полям осуществляется по имени при помощи селектора записи согласно следующему формату:

имя_переменной.имя_поля

Kto_to.F_I_O – это значение строки с фамилией.

Kto_to.Telephone – это значение длинного целого с заданным телефоном.

При присвоении первоначальных значений данным типа «запись» делают так:

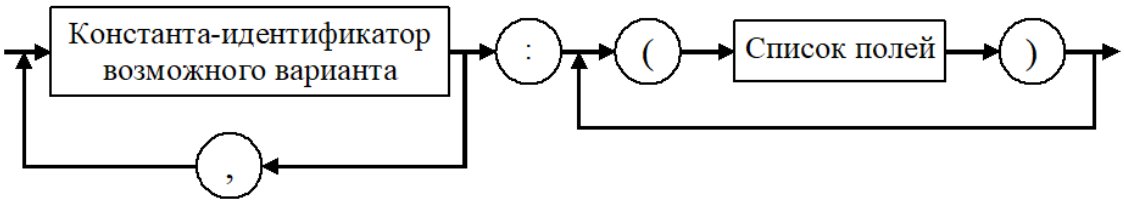
```
const
       Nekto : person=
                     (F_I_O  : 'XXX';     Ves : 50.5;
                      Rost   : 158; Telephone : 265539;
                      Ozenki : (5,4,3,4));
```

Поле от своего значения отделяется символом «:». Порядок чередования полей должен соответствовать порядку их описания в типе, поля должны разделяться НЕ запятой, а «;», как это делается в описании типа «запись».

Дискриминант позволяет контролировать включение в запись тех или иных вариантов.

Если на схеме выбран путь через идентификатор поля признака далее к идентификатору типа, то поле в записи присутствует, иначе – отсутствует.

Альтернатива:



Константа-идентификатор возможного варианта принадлежит типу поля признака и дает значение поля признака.

Из определения типа запись видно, что запись содержит фиксированное количество компонент, которые называются *полями*. Количество полей в записи, а также их назначение известны заранее. В отличие от компонент массива (выбираемых по своим индексам) доступ к полям записи осуществляется через имена полей. Имя поля должно быть уникальным только в пределах данной записи. Размер памяти, необходимый для записи, рассчитывается исходя из длин полей (если с вариантной частью, то максимальное поле переменной части). Функция `SizeOf(Z)` – возвращает длину записи `Z`.

Список полей может иметь только одну вариантную часть, которая всегда располагается после фиксированной, но вариантная часть сама может содержать и фиксированную часть и вариантную. Это видно из схемы.

Примером такой структуры данных может служить анкета служащего какого-то предприятия (фактически записи и были введены для описания таких данных), которая содержит следующие поля: *табельный номер; фамилия, имя, отчество; пол; код специальности; оклад*.

Переменная в программе, которая предназначена для хранения данных этой анкеты, может быть определена так:

```
var
    Person: record
               Tab_num    : 0..9999;
               Name       : String[20];
               SecondName : String[20];
               SurName    : String[20];
               Sex        : (male, female);
               Special    : 0..99;
               Oklad      : Real;
            end;
```

Однако, программист может наперед определить новый тип данных – запись `TPerson`, а затем описывать переменные с введенным типом.

```
type
    TPerson = record
               Tab_num    : 0..9999;
               Name       : String[20];
               SecondName : String[20];
               SurName    : String[20];
```

```
                Sex        : (male, female);
                Special    : 0..99;
                Oklad      : Real;
              end;
var
    Person1, Person2 : Tperson;
```

Над записями как единым целым допускаются такие действия: присвоить значение одной записи другой, проверить их на равенство или неравенство. Но такие записи должны иметь одинаковый тип. Ввод-вывод записи происходит только по их полям. Доступ к элементам (полям) записей происходит при помощи конструкции, которая называется *селектор записи*, имеющей следующий вид:

```
R.f
```

где `R` – переменная типа запись, `f` – идентификатор поля.

Для переменных `Person1` и `Person2`, введенных выше, допустимы следующие присваивания:

```
Person1.Name    := 'Александр';
Person2.Name    := 'Мария';
Person1.Sex     := Male;
Person2.Sex     := female;
Person1.Special := 12;
Person2.Special := Person1.Special;
```

Комбинированные типы могут использоваться для построения более сложных структур, например, массивов, или записей, содержащих поля-записи:

```
var
    Group    : array [1 .. 10] of TPerson;
    DataBase : file of TPerson;
```

Доступ к полям записей, образующих массив, осуществляется таким образом:

```
Group[j].Sex := female;
…
if Group[j].Name='Борис' then
                          Writeln (Group[j].Surname);
```

Пример 3. Пусть для комбинированного типа TPerson необходимо хранить информацию о дате рождения человека и дате поступления на работу.

Такая информация могла бы быть представлена в виде трех полей в самой записи `TPerson`. Однако это нерационально. Здесь логично определить отдельный тип даты, и тогда его можно использовать в описании других типов и переменных:

```
type
      Date = record
               Month : 0..12;
               Day   : 1 .. 31;
               Year  : 1900..2050
             end;
```

Введенный таким образом тип далее можно использовать в записи TPerson:

```
type
     TPerson = record
                 Name, SecondName, SurName : String [20];
                 Sex        : (Male, Female);
                 Speciality : Word;
                 BirthDay   : Date;
                 WorkDay    : Date
               end;
    var     Person1, Person2 : TPerson;
```

Доступ к внутренним полям поля Birthday осуществляется по общим правилам:

```
Person1.BirthDay.Day  := 12;
Person1.BirthDay.Year := 1980;
Person1.WorkDay.Year  := 1999;
```

Записи с вариантами

Понятие записи в языке Pascal предусматривает одну не совсем тривиальную возможность: в пределах одной записи иметь разную информацию в зависимости от конкретного значения некоторого поля – дискриминанта. Непосредственный выбор структуры записи будет определяться значением поля дискриминанта. Вариантные поля описываются после фиксированных полей и оформляются иным образом.

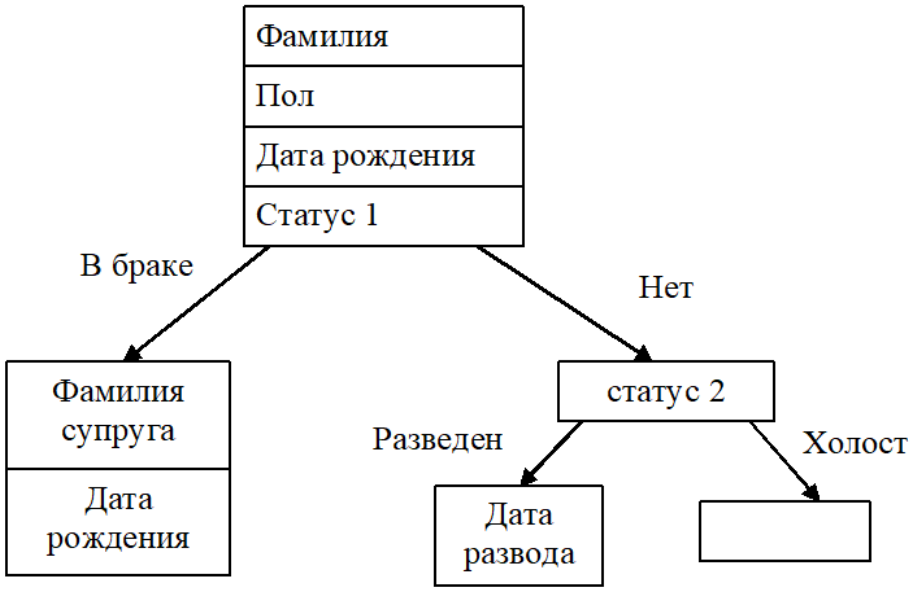
Важно понимать, что в любое время доступны поля только одного из всех возможных вариантов, описанных в вариантной части записи. Все варианты при хранении распределяются на одном и том же месте памяти, а размер этого места определяется самым объемным из вариантов.

Результатом такого способа хранения вариантов является опасность использования полей не того варианта, который соответствует текущему значению поля дискриминанта. Значит, при присваивании начального значения записи нужно сначала задать значение компонента дискриминанта, а затем – компонентам вариантной части. Pascal не содержит никаких средств контроля за правильностью работы с вариантами записей.

Когда в вариантной части само поле признака отсутствует, а присутствует имя порядкового типа, то мы можем обращаться к любой альтернативе в вариантной части. Это используют тогда, когда необходимо обеспечить разнотипное представление одних и тех же данных.

Задача. Пусть требуется хранить следующие сведения о сотрудниках института: фамилия, пол, дата рождения; если женат/замужем, тогда фамилия и дата рождения супруга; если холост, но был в браке ранее, – сведения о дате развода. Описать тип данных, необходимый для хранения указанной информации.

Структура такой анкеты следующая:



Если бы не было вариантной части, то надо было бы описать три разные типы анкеты. Обслуживать же один тип анкеты в программе – проще.

Решение.

```
program PR_Zapis;
type
      MarStatus = (married, Single, divorced, no);
      Data = record
             Month : 1..12;
             Day   : 1 .. 31;
             Year  : 1900 ..2050
             end;
      Man  = record
             Surname    : String [20];
             Sex        : (Male, Female);
             Birthday   : Data;
             case yesno : MarStatus of
                married :
                        (Name: String[20]; MarDay : data);
                 Single :
                          ( case yn : MarStatus  of
                           divorced: (DivDay:data);
                           no      :  (  )
                           )
             end;
 var      Person3, Person4 : Man;
```

В данном типе `Man` фиксированные поля: `SurName, Sex, Birthday, yesno`. *Первая* вариантная часть содержит две альтернативы – `married, Single`, состоящие из списка полей. Альтернатива `married` имеет фиксированные поля:

Name, MarDay; у альтернативы Single нет фиксированной части, а есть вариантная часть. *Вторая* вариантная часть содержит две альтернативы – divorced и no. Альтернатива divorced имеет фиксированное поле DivDay; альтернатива No вообще не имеет полей.

Когда вводятся сведения о сотруднике, который находится в браке, то запись будет содержать следующие поля (таблицы 19-21):

Таблица 19 – Первый пример записи.

SurName (String [20])	Sex (Male, Female)	Birthday (data)			Yesno (Mar Status)	Name (String [20])	Marday (data)		
Иванов	Male	6	31	1970	Married	Краснова	2	28	1991

Если сотрудник холост и не разведен, тогда запись будет содержать такие поля:

Таблица 20 – Второй пример записи.

SurName	Sex	Birthday			Yesno	yn	
Сидорова	Female	11	2	1960	Single	No	(пусто)

Если одинокий и в разводе, тогда так:

Таблица 21 – Третий пример записи.

SurName	Sex	Birthday			Yesno	yn	divDay		
Кольцов	Male	12	31	1961	Single	Divorced	5	5	1993

Пример записи с вариантами без поля-признака.

```
type
      CharArrayType = array [1 .. 4] of Char;
var
      V4 : record
           case Boolean of
                true  : (C : CharArrayType);
                false : (b1, b2, b3, b4 : Byte)
           end;
```

Размер переменной V4 – 4 байта. Обращение к V4.C – это обращение к массиву из четырех символов; обращение V4.C[2] – ко второму элементу этого массива; V4.B1, V4.B2, ... – это числовое значение символа (ASCII – код элементов символьного массива).

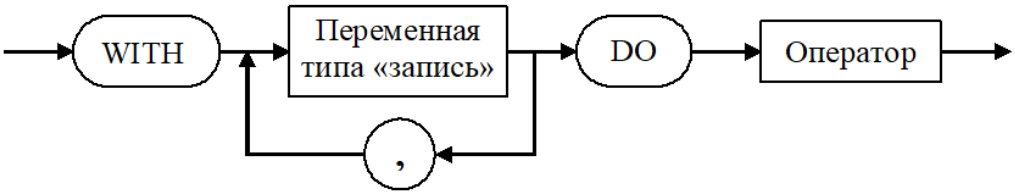
Переменная типа «запись» может участвовать только в операциях «:=» или операциях обмена для файлов с типом. А поля записи могут принимать участие во всех операциях, которые допускает тип поля.

Оператор присоединения with

При работе с переменными-записями часто приходится много раз выписывать одни и те же имена, и обращение к полям записи имеет несколько громоздкий вид, так как образуется составное имя при помощи селектора записи.

Для сокращения записи в языке существует так называемый оператор присоединения **with**. Он выделяет в тексте программы область, в которой действует общая часть имени переменной-записи.

Синтаксис оператора WITH:



Внутри оператора, который стоит после **do**, к полям переменных, перечисленных в списке имен записи, можно обращаться как к обычным переменным.

Pascal допускает вложение записей одна в другую (это значит поле в записи может быть в свою очередь записью). Соответственно оператор **with**, в свою очередь, может быть вложенным (распространение оператора **with** на несколько полей в глубину):

```
with RV1 do
     with RV2 do
               …
          with RVn do …
```

что эквивалентно записи

```
with RV1, RV2, … , RVn do …
```

Уровень вложенности ограничен.

Необходимо помнить, что все идентификаторы в пределах оператора присоединения проверяются, можно ли их интерпретировать как поля записи, упомянутые в заголовке. Когда это так, то они всегда понимаются таким образом, даже когда в текущем блоке доступны и переменные с такими же идентификаторами.

Имена полей как бы локализованы внутри записи, это значит они могут совпадать с именами других переменных.

Например, определение

```
var  a : array [2..8] of Integer;
     a : 1..5;
```

недопустимо из-за двусмысленности a.

А следующее определение допустимо:

```
var
```

```
        a : Integer;
        b : record
            a : Real;
            b : Boolean
            end;
```

Проанализируем два варианта следующей программы, в которых показано, как можно обратиться к полям записи, и к переменной с одноименным идентификатором.

Вариант 1:

```
program Main;
var
     x, y : Integer;
    RecXY : record
             x, y : Real;
            end;
begin
    x       := 10;
    y       := 20;
    RecXY.x := x*3.14;          {здесь селектор записи}
    RecXY.y := y*3.14;
     …
end.
```

Вариант 2:

```
program Main;
var
     x, y : Integer;
    RecXY : record
             x, y : Real;
            end;
begin
       X := 10;
       Y := 20;
       with  RecXY  do
             begin
           X := 3.14*Main.X;
               Y := 3.14*Main.Y
            end;
          …
end.
```

Использование записи с вариантами

Рассмотрим пример использования записи с вариантной частью не в традиционных задачах обработки анкетных данных.

Задача. В заданной матрице A_{nm} (n, m – константы), рассматривая ее как совокупность элементов размера $n \times m$, в каждой строке отсортировать элементы в порядке возрастания, причем сделать так, чтобы в строке с меньшим номером были элементы не большие, чем элементы в строке с большим номером.

Решение.

Наложим матрицу на вектор, сделав запись с вариантами, и выполним сортировку поля-вектора. Потом распечатаем поле-матрицу.

```
program MatrVect;
const       n = 7;  m = 8;
type Matr = array [1 .. n, 1 .. m] of Real;
     Vect = array [1 .. n * m]     of Real;
     TRec = record
            case Byte of
               0 : (A : Matr);
               1 : (B : Vect)
            end;              {наложили матрицу на вектор}
var      P    : TRec;
         r    : Real;
         i, j : Integer;
begin
with P do
     begin
     Writeln ('A – ?');  {ввод матрицы}
     for i := 1 to n * m do
         Read (B [i]);
     Writeln;
     Writeln ('МАТРИЦА НА ВХОДЕ ');
     for i := 1 to n do
         begin
         for j:=1 to m do
             Write (A [I, J]:6:0);
         Writeln;
         end;
     for i := 1 to n * m - 1 do  {Сортировка методом пузырька}
         for j := i + 1 to n * m do
             if B[i] < B[j] then         {в порядке убывания}
                 begin
                  r    := B[i];
                  B[i] := B[j];
                  B[j] := r;
                 end;
     Writeln;   Writeln ('МАТРИЦА НА ВЫХОДЕ');
     for i:=1 to n do
         begin
          for j:=1 to m do
             Write (A [I, J]:6:0);
```

```
          Writeln;
         end;
     end;  {with}
end.
```

Изменение (приведение) типов и значений

В языке Pascal существует мощное средство, позволяющее обойти все возможные ограничения на совместимость типов или значений (например, в операторе присваивания «:=»). Эта *операция приведения типов*. Она применяется *только* к переменным и значениям.

Суть этой операции в следующем. При определении типа мы определяем форму хранения информации в оперативной памяти, и переменная данного типа будет представлена в памяти наперед известной структурой. Но если «посмотреть» на ее образ в памяти с точки зрения машинного представления другого типа, тогда можно будет трактовать то же самое значение как значение, принадлежащее к другому типу.

Формат операции приведения типов:

```
Имя_типа (переменная_или_значение);
```

Это `имя_типа` должно быть известно в программе (указано программистом или стандартное).

Приведение типов НЕ переопределяет типы переменных, а только дает возможность нарушить правила совместимости типов при условии, что соответствующие значения совместимы в машинном представлении.

Пример.

```
Var     Si : 0..255;
```

Используя оператор `Si:='A'` – получим ошибку на этапе компиляции. А оператор `Char(Si):='A'` – позволит сделать присваивание значения.

Возможен и другой подход: `Si:=Ord('A')`, в этом случае результат возвращается как `Byte`.

Аналогично изменению типа переменных можно изменять собственно тип значений, а также итоговый тип выражений, т.е. разрешены такие преобразования: поля (таблица 22):

Таблица 22 – Приведение типов данных.

`Boolean(1)`	Это логическое значение **`true`**
`Longint(1)`	Это единица, размещенная в 4 байтах
`Char(130-1)`	Символ с ASCII кодом номера 129
`Integer('Y')`	Код символа `'Y'` в формате `Integer`, {`Ord('Y')` возвращает результат как `Byte` }

В результате изменения (приведении) типа данных может измениться объем памяти, требуемый для хранения приведенных переменных и значений по сравнению с первоначальным типом данных.

Приведение типов, как переменных, так и значений – нетривиальная операция. Она требует довольно высокого уровня знаний технических подробностей языка.

Нужно знать, как хранятся в памяти сложные типы данных (массивы, записи, множества), адреса, числа, строки, сколько байт памяти им отводится.

Приведение типов имеет смысл только при сопоставимости машинных представлений значений.

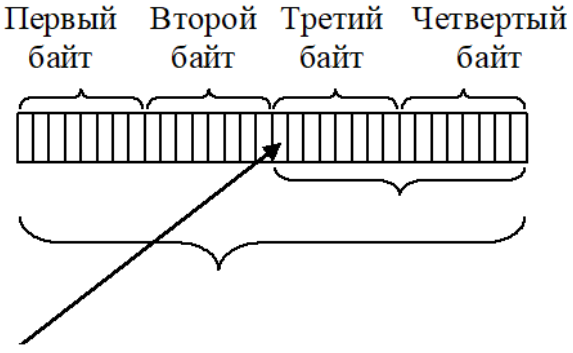
Вспомним, что целые и действительные значения имеют совершенно различное машинное кодирование.

Значит, приведение целых переменных к вещественным типам и наоборот – очень рискованная операция.

Когда один порядковый тип превращается в другой, такое преобразование может привести к усечению или увеличению размеров памяти, по сравнению с исходным значением. При этом можно получить «плохие» ответы:

```
var   li : Longint;  i : Integer;
begin
     li := 1234567;        {>32767}
     I  := Integer(li+1);
 Writeln(I);
end.
```

Первоначально в переменной `li` имеем: $(1234567)_{10}=(12D687)_{16} \Rightarrow$ `Longint`. Переменная `i` получит значение двух последних байт, при этом старший бит этого поля содержит значение 1, т.е. переменная `i` будет иметь отрицательное значение и напечатается –10616. Это хорошо видно из следующей схемы.



Замечание 1**.** При приведении значения в более широкий тип (например, Longint (1)) значения будут полностью записаны в младшие байты, если же значение приводится к более короткому типу, от него берутся опять же младшие байты (а старшие игнорируются). В таком случае приведенное значение может не равняться исходному.

Пример. Результат выполнения `Byte(534)`будет равен `22`.

534 – кодируется в тип `Word` как $2 \cdot 16^2 + 22 = 2 \cdot 16^2 + 16 + 6$. Значит, $534_{10} = 216_{16}$

0	2	1	6

Младший байт $(16)_{16} = 16 + 6 = 22$ мы получим, а старший – основной – потерялся.

Замечание 2**.** Если исходное значение отрицательное, а заданный тип расширяет размер его хранения, тогда знак будет сохраняться.

На практике надо пользоваться операцией приведения типов тогда, например, когда работаем на границе значений для данного типа.

```
var    A, B : Word;
  begin
       A := 55000;
       B := A-256;
       Write ( A + B ); …
```

Так как `A + B > 65535`, а `A + B` дает тип выражения `Word`, значит, результат действия `Word(A+B)` заведомо плохой. Исправим эту ситуацию так:

```
Write(Longint(A)+B).
```

Задача. Что напечатается в следующей программе?

```
type
      Arr4 = array[1..4] of Byte;
var   x    : Longint;
begin
  Arr4(x)[1] := 6;     Arr4(x)[2] := 7;
  Arr4(x)[3] := 1;     Arr4(x)[4] := 0;
  Writeln(x);   Readln;
end.
```

Решение.

Младший байт переменной `x` типа `Longint` стал равен 06, следующий – 07, следующий – 01 и старший – 00.

Запишем это шестнадцатеричное число в привычном виде: $x = 00\ 01\ 07\ 06_{16}$.

Переведем его в десятичную систему счисления: $x = 1 \cdot 16^4 + 7 \cdot 16^2 + 6 = 67334_{10}$.

Напечатается 67334.

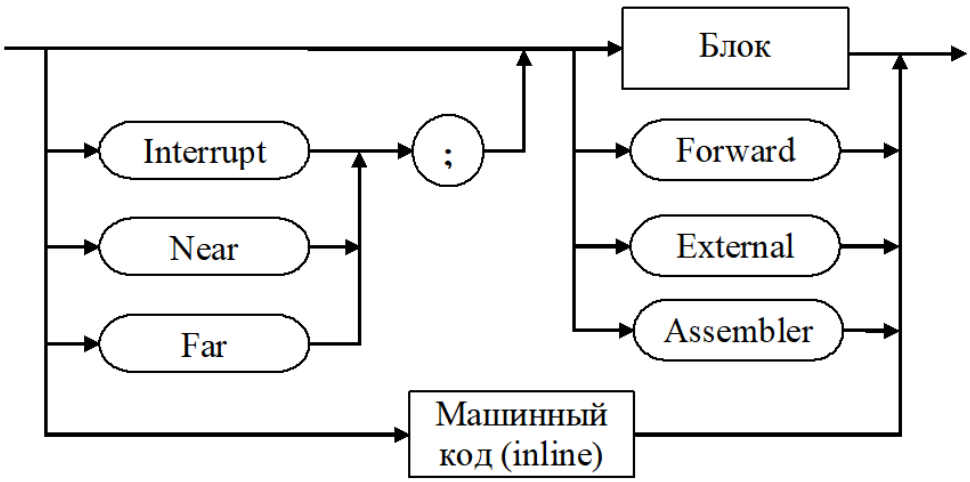
1.11. Процедуры и функции

Содержание

- Механизмы структурирования программ.
- Процедуры пользователя.
- Функции пользователя.
- Параметры.

Механизмы структурирования программ

В разделе «Простейшее определение процедур и функций» введены понятия заголовок процедуры и тело процедуры в простейшем виде. В общем виде тело процедуры определяется следующей синтаксической диаграммой:



Синтаксическую диаграмму «Блока» смотрите в теме «Общая структура Pascal программы».

Опция или директива `INTERRUPT` служит для определения так называемых процедур прерываний, когда в программе необходимо определить собственные алгоритмы реакции на прерывания операционной системы, убрав при этом стандартные реакции. Это процедуры специального вида.

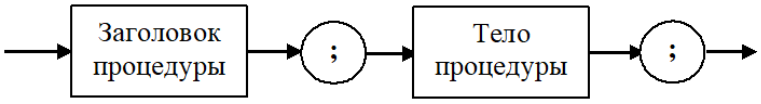
В языке Pascal, начиная с версий выше 4.0, интерфейс связи с языком ассемблер принял завершенный вид: язык позволяет задавать тело подпрограммы непосредственно в виде последовательности машинных команд, используя специальную конструкцию с опцией **`inline`**, или в виде серии ассемблерных инструкций, с использованием опции `ASSEMBLER`.

Опции `NEAR` или `FAR` определяют, как компилятору строить машинное представление подпрограммы: ближний тип вызова – `NEAR`, дальний тип вызова – `FAR`. Эти опции применяются в основном при использовании подпрограмм в качества формальных параметров для данных процедурных типов.

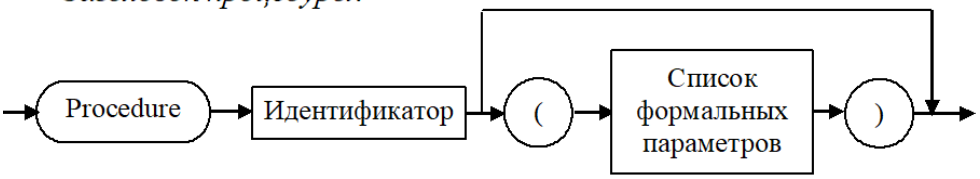
Опция `EXTERNAL` используется тогда, когда надо обратиться к подпрограмме, тело которой написано вне вашей программы и находится в виде объектного кода (было ли записано в машинных командах, после ли трансляции с какого-то алгоритмического языка). Это так называемое внешнее описание.

Процедуры пользователя

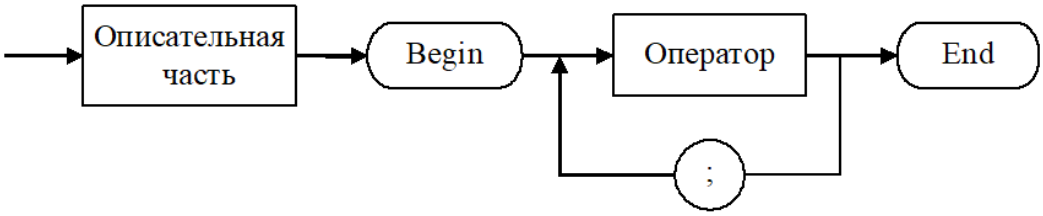
Синтаксическая диаграмма простейшего определения процедуры:



Заголовок процедуры:



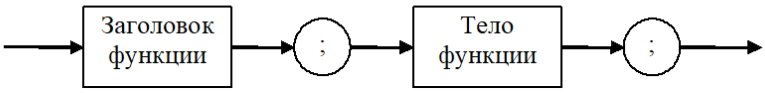
Тело процедуры в простейшем виде определяется следующей синтаксической диаграммой:



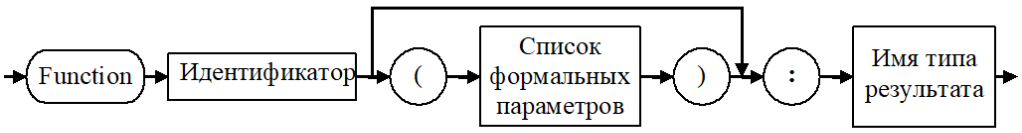
Функции пользователя

Функция, в отличие от процедуры, подсчитывает только одно простое скалярное или строковое значение, которое можно использовать в том или ином выражении.

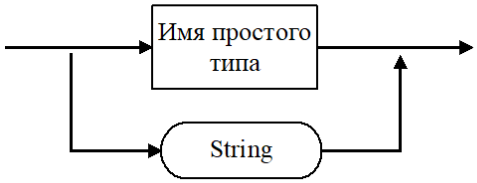
Синтаксическая диаграмма простейшего определения функции:



Заголовок функции:



Имя типа результата или тип функции:



Тело функции аналогично телу процедуры, за исключением того, что отсутствует опция `INTERRUPT`.

В теле функции в разделе операторов должен находиться, по меньшей мере, один оператор, который присваивает идентификатору функции результирующее значение.

Каждая процедура или функция может иметь свои процедуры или функции – локальные.

Параметры

Параметры обеспечивают механизм замены, который позволяет выполнять подпрограммы с разными данными.

Определение типа параметра (стр.84) накладывает ограничения на запись типа параметра. Например, следующее описание заголовка процедуры не корректно:

```
procedure InCorr(var A : array[1..10, 1..5] of Byte);
```

Тип массива здесь задается конструкцией описания массива, а не именем типа, и эта ошибка будет замечена на этапе компиляции.

Тип массива нужно описать заранее в секции **type** и затем использовать его имя, например, так

```
type mas = array[1..10, 1..5] of Byte;
procedure InCorr(var A : mas);
```

Параметры ”открытые массивы“

Если в подпрограмме параметры-массивы желательно использовать для одномерных массивов разной длины, можно пользоваться параметрами – открытыми массивами, в которых указывается базовый тип элементов массива, но не очерчиваются его размеры и границы.

Конструкция

```
array of тип_элементов
```

задает так называемый *открытый массив*.

Например,

```
procedure InCorr(aArray : array of Integer);
```

Массив `aArray` только одномерный, в теле подпрограммы индекс его всегда начинается с нуля и изменяется до величины `High(aArray)`.

`High(x)` – возвращает максимальное значение типа-диапазона, к которому принадлежит переменная `x`.

Пример 1.

```
const
      A : array[-1..2] of Integer =
                          (0, 1, 2, 3);
      B : array[ 5..7] of Integer =
                          (4, 5, 6);
procedure ArrayPrint(aArr : array of Integer);
 var
```

```
          k : Integer;
     begin
      for k := 0 to High(aArr) do
         Write(aArr[k]:8);
      Writeln;
      end;
    begin
       ArrayPrint(A);
       ArrayPrint(B);
    end.
```

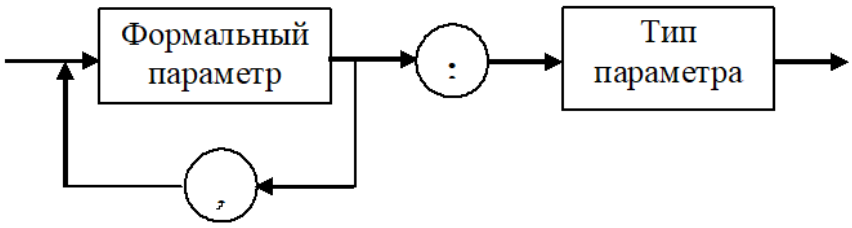
Рассмотрим на следующем примере как таким механизмом можно передать в качестве фактического параметра двумерный массив.

Пример 2.

```
type
   mas= array [1..2] of Shortint;
const
  a:array[1..5]  of mas =
                ((1,1),(2,2),(3,3),(4,4),(5,5));
  b:array[-1..1] of mas =
                ((-1,-1),(0,0),(1,1));
procedure Wr_Arr(const x : array of mas; c:Char);
      var
         i : Byte;
      begin
         Writeln(c:5);
         for i := 0  to  high(x) do
            Writeln(x[i][1]:4, x[i][2]:4);
         Writeln;
      end;
begin
   Wr_arr(a,'a');
   Wr_arr(b,'b');
end.
```

Параметры-значения

Синтаксическая диаграмма параметров-значений.



Параметр-значение является локальной (внутренней) переменной подпрограммы, которая существует только в границах определения подпрограммы.

Изменение параметра-значения в подпрограмме никак не влияет на значения фактического (соответствующего ему) параметра, ведь в ситуации работы с параметром-значением, компилятор строит какую-то свою переменную в памяти, пересылает в нее значение фактического параметра и работает с ней.

Когда подпрограмма заканчивает свою работу, доступ к такой переменной теряется.

Можно сделать следующие выводы.

Фактический параметр в данном случае может быть выражением (в частности константой или переменной).

Параметры-значения служат для того, чтобы подпрограмме передать какие-то сведения (фактически копии их), и их нельзя использовать, чтобы получить из подпрограммы какие-то результаты.

Параметры-переменные

Параметры-переменные используются, когда подпрограмма должна вернуть в вызывающую ее программу некоторые результаты.

Компилятор строит программный код так, что такие фактические параметры обрабатываются на своем месте в памяти (при помощи ссылок). Изменения значений таких параметров в подпрограмме становятся известными в программе, которая вызывает эту подпрограмму. При активизации (вызове) подпрограммы на месте фактических параметров-переменных передаются адреса их нахождения в памяти.

Предусматривая защиту от ошибочного изменения данных в программе, вызывающей подпрограмму, нужно использовать параметры-значения или же параметры-константы; если же в программу нужно передать результат – тогда параметры-переменные.

Пример. Напечатать в двоичном представлении некоторое целое число.

```
function Binary (x : Longint; NumOfbits : Byte) : String;
  {где Longint – 4 байта = 32 бита,
        а NumOfbits – кол-во реальных бит}
   var   bit, I : Byte;
         S      : String[32];
   begin
```

```
    S := '';
    for i := 0 to 31 do
      begin
       bit := (X shl i) shr 31;
       S := S + chr(Ord('0') + bit);
      end;
    Delete (S, 1, 32 - NumOfbits);
    Binary := s;
   end;   {конец функции}
var
     x, y, z : Integer;
begin
     x:=-32768;
     Writeln(Binary(x,16));
     y:=-(x shr 31);
     Writeln(Binary(y,16));
     y:=-y;
     Writeln(Binary(y,16));
     z:=x xor y;
     Writeln(Binary(z,16));
     y:=z-y;
     Writeln(Binary(y,16));
end.
```

Задание. Целое положительное число *m* записывается в двоичной системе счисления, и разряды в этом представлении переставляются в обратном порядке (реверс). Взять полученное число в качестве значения функции `Bit_Reveres(m)`. Напечатать значения `Bit_Reveres(m)` от *m*=512 до 1012 з шагом 50.

Еще один механизм передачи сведений подпрограммам основывается на принципе локализации.

Принцип локализации

Все метки, константы, типы, переменные, процедуры, функции, описанные в программе, называются *глобальными объектами* программы. Глобальные объекты доступны внутри подпрограмм, описанных в этой программе.

Все метки, константы, типы, переменные, процедуры, функции, которые описываются после заголовка подпрограммы, и параметры-значения называются *локальными объектами* подпрограммы. Локальные объекты доступны только в пределах этой подпрограммы, включая вложенные подпрограммы, но недоступны вызывающей (охватывающей) программе (подпрограмме). Локальные объекты охватывающей подпрограммы являются глобальными по отношению к описанными в ней подпрограммам.

Локальные объекты создаются при входе в подпрограмму и уничтожаются при выходе из нее. Если имя объекта определено в нескольких подпрограммах, то в каждой подпрограмме этому имени соответствует свой локальный объект.

Локализация переменных обеспечивает свободу в выборе идентификаторов. Так, если две подпрограммы отделены друг от друга (т.е. не вложены одна в другую), то идентификаторы в них могут быть выбраны совершенно произвольно, в частности, могут повторяться.

В случае совпадения идентификаторов переменных им соответствуют разные области памяти, совершенно не связанные друг с другом.

Если некоторое имя определено и в программе, и в вызываемой ею подпрограмме, то внутри подпрограммы глобальный объект недоступен, он как бы экранируется локальным объектом с таким же именем.

Изменение в подпрограмме локального объекта никак не проявляется в охватывающей программе (подпрограмме).

Изменение глобального объекта – проявляется.

Значит, результаты, полученные в подпрограмме, можно использовать в охватывающей программе (подпрограмме) только тогда, когда присвоить их значения глобальным переменным (сюда входят и параметры-переменные).

Не рекомендуется использовать глобальные переменные внутри подпрограмм.

Изменение значений таких переменных в подпрограмме приводит к изменению их значений и в вызывающей программе, что не всегда желательно.

Кроме того, использование глобальных переменных в подпрограммах затрудняет перенос кода в другие программы.

В Турбо Паскале допускается любой уровень вложенности процедур и функций.

Любая процедура, описанная в основной программе, в свою очередь, может иметь описания внутренних процедур и функций и т.д. При этом объекты, описанные в вызывающей процедуре, являются глобальными по отношению к вызываемой процедуре.

Область действия меток переходов всегда локальная, поэтому нельзя планировать переходы из подпрограммы в охватывающей программу при помощи **goto**.

Пример. Какой будет результат работы программы?

```
program Pr;
var
       x, y : Integer;      {  x   y                }
 procedure Proc;
   var
          y : Byte;
    begin
       x := x + 1;
       y := 1;
       Writeln ('в подпрограмме x=', x : 4, ' y=', y : 4);
    end;
      { в подпрограмме  x=1 y=1}
var
     a, b, c : …;
```

```
    {эти глобальные переменные уже недоступны в Proc}
begin
    x := 0;
    y := 0;
    Proc;
    {активизируется подпрограмма Proc, в которой
     глобальная переменная x получит другое значение}
    Writeln ('в программе x=', x : 5, ' y=', y : 5);
                                    {в программе x=1 y=0}
end.
```

Побочный эффект

Если при вызове подпрограммы кроме изменения параметров-переменных происходят изменения каких-то глобальных объектов, то это называется *побочным эффектом*.

Побочный эффект считается нежелательным явлением, так как он может привести к неконтролируемости ситуации.

Особенно вреден побочный эффект для функции.

Основное назначение функции – по ее параметрам получить результат и явно его вернуть.

Если при этом изменяются глобальные переменные или параметры-переменные, тогда содержание функции становится неясным.

Пример 1.

```
program SideEffect;
var
    k : Integer;
function Fact (var n : Integer) : Longint;
   var
       f : Longint;
  begin
   f:=1;
   repeat
         f := f * n;
         n := n - 1;
   until n = 1;
   Fact := f
end;

function Fk(k : Integer) : Integer;
   begin
     Fk := k;
   end;
begin
 k := 3;
 Writeln (k, '  ', Fk(k) + Fact(k));
```

```
 k := 3;
  Writeln (k, '  ', Fact(k) + Fk(k));
end.
```

Наша подпрограмма `Fact` изменяет параметр-переменную. По причине этого побочного эффекта от перемены мест слагаемых изменился результат.

Чтобы убрать этот побочный эффект, нужно вместо параметра-переменной «`var n : Integer`» описать параметр-значение «`n : Integer`».

Многие современные языки программирования содержат прямые запреты на подобные действия.

Пример 2.

```
program SideEffect_2;
var
    a, d, z : Integer;
function Change(x : Integer) : Integer;
  begin
   z := z - x;           {изменение глобальной переменной}
   Change := Sqr(x);     {аргумент в квадрат             }
  end;
begin
 z := 10;
 a := Change(z);
 Writeln(a, z);     {100, 0}
 z := 10;
 d := 10;
 a := Change(d)*Change(z);
 Writeln(a, z);
 z := 10;
 d := 10;
 a := Change(z)*Change(d);
 Writeln(a, z);
end.
```

Выполним программу на компьютере и сравним ответы. Видно, что при вычислении значения переменной *a* изменение последовательности вызовов функций `Change(d)` и `Change(z)` с побочным эффектом привело к разным ответам.

Замечание 1. Нужно избегать зависимости подпрограмм от глобальных в отношении нее переменных.

1.12. Рекурсия и итерация

Содержание

- Примеры рекурсивных подпрограмм.
- Параметры без типа.

Примеры рекурсивных подпрограмм

Рекурсия – способ организации вычислительного процесса, при котором процедура или функция в процессе выполнения входящих в ее состав операторов обращается сама к себе.

Для большинства вычислительных задач итерация – последовательное вычисление – оказывается эффективнее, чем рекурсия, но для многих задач, которые работают с данными сложной структуры, использование рекурсии является предпочтительным.

В теле подпрограммы известны (доступны) все объекты, которые описаны в охватывающем ее блоке, в том числе и имя самой подпрограммы. Таким образом, внутри тела подпрограммы возможен вызов самой подпрограммы.

Процедуры и функции, которые используют вызовы «самих себя», называются *рекурсивными*.

Допустима также косвенная рекурсия, при которой, например, подпрограмма A вызывает B, которая в свою очередь вызывает подпрограмму C, а последняя – первоначальную подпрограмму A.

Хорошо известно, что существует много рекурсивных математических алгоритмов, например вычисление $n!$:

$$n! = (n-1)! \cdot n, \quad 0! = 1, \quad 1! = 1.$$

```
function Fact(N : Byte): Longint;
begin
 if N = 0 then Fact := 1 else Fact := N * Fact(N - 1);
end;
```

Выполнение рекурсивной подпрограммы происходит в два этапа:

- На первом этапе осуществляется построение рекурсивных соотношений и частичное вычисление с задержкой выполнения действий, которые не могут быть выполнены на данном этапе.

Задержка вычислений обусловлена тем, что в описании рекурсивной подпрограммы всегда присутствует обращение к той же подпрограмме.

Первый этап завершается после выполнения так называемого условия завершения рекурсии. (Это дно рекурсии.)

- На другом этапе осуществляются вычисления действий на основании рекурсивных соотношений.

В языке Pascal нет никаких ограничений на рекурсивные вызовы подпрограмм, необходимо только хорошо понимать, что каждый очередной рекурсивный вызов приводит к образованию новой копии локальных объектов подпрограммы, и все эти копии, соответствующие цепочке активизированных и незавершенных рекурсивных вызовов, существуют независимо друг от друга в оперативной памяти.

Обычно рекурсивные подпрограммы применяют при небольшой глубине рекурсии, когда время счета и затраты памяти не очень большие.

Программируя рекурсию, всегда нужно помнить об условии ее окончания, так как в противном случае подпрограмма будет бесконечно обращаться сама к

себе и теоретически никогда не остановится, но, скорее всего, программа исчерпает ресурсы ПК, поскольку каждое обращение к подпрограмме требует очередной порции оперативной памяти.

Нужно уметь различать, где рекурсия, а где итерация.

Следующие алгоритмы можно рассматривать и как итерационные и как рекурсивные.

Задача 1. Подсчитать N-е число Фибоначчи: $f_0=1$, $f_1=1$, $f_n=f_{n-1}+f_{n-2}$, $n\geq 2$.

```
function ChFR(n : Integer) : Integer;
begin
   if (n = 0) or (n = 1) then ChFR := 1
   else ChFR := ChFR(n - 2) + ChFR(n - 1)
end;
```

Задача 2. Найти НОД(a, b) (НОД – наибольший общий делитель).

```
function HAD(a,b : Integer) : Integer;
begin
 if b = 0 then HAD := a  else HAD := HAD(b, a mod b)
end;
```

Задача 3. Подсчитать значение суммы элементов

$$S_n = \sum_{k=1}^{n} a_k = S_{n-1} + a_n, \quad S_1 = a_1.$$

```
const k=50;
type    AR = array[1..k] of Real;
var      a : AR;
         s : Real;
      n, i : Integer;
  function Sum(var a : AR; n : Integer) : Real;
  begin
   if n = 1 then Sum := a[1]
            else Sum := Sum(a, n - 1) + a[n]
  end;
begin
 n := 10;
 for i := 1 to n do
     a[i] := Random * 100 - 50;
 S := Sum(a, n);
 Write('сума=', S);
end.
```

Задача 4. Найти наибольшее значение среди чисел $a_1,\ldots,a_n$.

Алгоритм. Сводим поиск до Max{a, b}. Имеем:

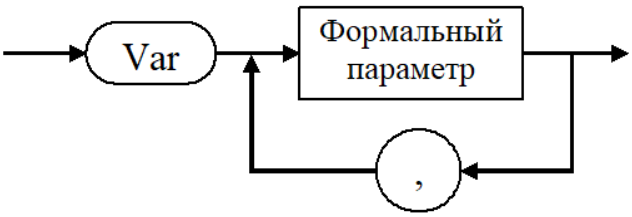
Max{a_1, …, a_n}=Max{Max{a_1, …, a_{n-1}}, a_n}, Max{a_1}=a_1

```
const n=15;
```

```
type      Vect=array[1..n] of Real;
var   i : Integer;
      a : Vect;
  function Maximum(var a : Vect; n : Integer) : Real;
  var M : Real;
  begin
   if n=1 then Maximum:=a[1]
   else
    begin
     M:=Maximum(a, n-1);
     if M<=a[n]  then   Maximum:=a[n]
                 else   Maximum:=M
     end;
  end;
begin
 for i := 1 to n do Read(a[i]);
 Writeln;
 Writeln(Maximum(a, n):9);
end.
```

Параметры без типа

Синтаксическая диаграмма параметров без типа:



Передача параметров без типа означает передачу в подпрограмму только адреса объекта. На этом же месте может появиться переменная любого типа.

Использовать такие параметры надо с осторожностью, так как в таких случаях язык Pascal не обрабатывает возможных ошибочных ситуаций несоответствия типа данных.

Если вместо данного конкретного типа передается его местоположение в памяти и длина в байтах, то с ним можно работать, как с данным нужного программисту типа путем приведения типов или наложения на такой параметр данного нужного типа.

Встроенная процедура `Move`, копирующая байты, параметрами которой являются переменные без типа:

```
procedure Move (var Source, Dest; Count : Word);
```

Копируется определенное число байт (`Count`) из `Source` в `Dest`. Проверка диапазона не выполняется.

Всегда, когда возможно, используйте `SizeOf`, чтобы определять количество копируемых байтов.

Пример. Распечатать значение переменной любого типа в 16-й с/с.

Алгоритм: Определим процедуру Dump от двух параметров: имя переменной и ее длина в байтах. Тогда решение будет такое:

```
program Write_Value;
var    x : Real;
       s : String;
       a : array[1..255] of Integer;
 procedure Dump(var z; len : Word};    {длина z в байтах}

     ...               {реализацию рассмотрим чуть позже}

     end;

    begin
     Dump(x, SizeOf(x));
     Dump(a, SizeOf(a));
                        {там случайная информация – мусор}
     x:=1;
     s:='0-x-0x-0x';
     Dump(x, SizeOf(x));
     Dump(s, SizeOf(s));
    end.
```

Теперь разработаем **procedure** Dump.

Алгоритм. На переданную переменную Z длиной len байт наложим массив байтов.

В каждом байте содержатся две 16-е цифры из множества базисных чисел: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

Каждый байт будем расшифровывать как строку из двух символов (каждый символ – 16-я цифра-символ) и потом печатать его.

Опишем для расшифровки функцию Hext.

```
procedure Dump(var Z; len : Word);
type
    St2 = String[2];
var
     M : array[1..$FFF0] of Byte ABSOLUTE Z;
      {это размещение массива по конкретному адресу
       в оперативной памяти (Absolute)}
      i : Word;

function Hext(B : Byte) : St2;
const
    HexDigit:array[0..15] of Char='0123456789ABCDEF';
   begin
    Hext:=HexDigit[B shr 4]+HexDigit[B and $F];
   end;
```

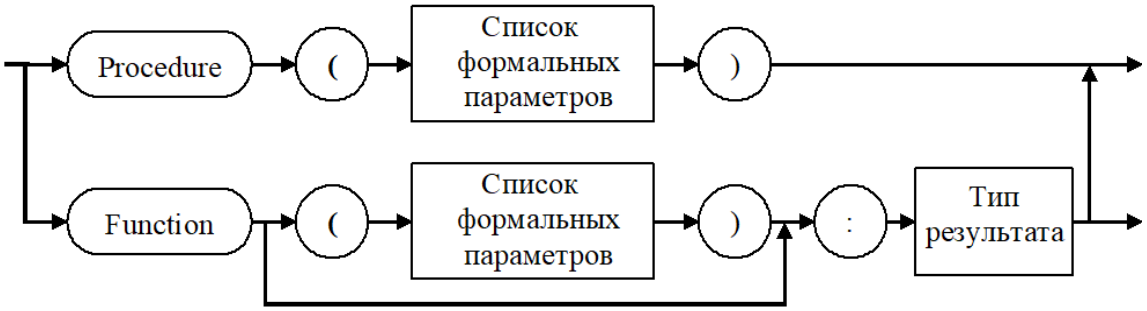
```
begin
 Writeln;
 Writeln;
 for i:=1 to len do Write(Hext(M[i]));
 Writeln
end;            {Dump}
```

Процедуры и функции как параметры

Отличительной особенностью языка Pascal является возможность передавать в подпрограмме имена других подпрограмм, оформляя их как параметры. И так же, как передавалось значение, может передаваться какая-то подпрограмма обработки данных. Особенно важным это становится при программной реализации алгоритмов вычислительной математики.

Определение процедурного или функционального типа аналогично заголовку подпрограммы с тем отличием, что имя подпрограммы не задается.

Процедурный тип:



Имена параметров могут быть выбраны произвольно.

Примеры.

```
type
 Proc         = procedure;
 BinOperation = function(x, y : Real) : Real;
 UnOperation  = function(x : Real) : Real;
 Reading      = procedure(var F:Text; var Elem:Char);
```

Процедурные типы также могут участвовать в построении составных типов.

Пример.

```
type
   Proc = procedure(T : Real);
var
 ArProc : array[1..10] of Proc;
```

К описанным выше данным можно обращаться следующим образом: `ArProc[7]((x+17.5)*2)`.

После объявления процедурного или функционального типа его можно использовать в описаниях параметров подпрограмм. И ясно, что необходимо

написать те реальные процедуры и функции, которые будут передаваться как параметры. Требование к ним одно: они должны компилироваться в режиме дальнего вызова (опция FAR или {$F+}).

Пример. Напечатать в 16 с/с таблицу сложения и умножения.

```
program Tables;
const
      HD : array[0..15] of Char = '0123456789ABCDEF';
type
     TFunc = function(a, b : Integer) : Integer;
     TSt2  = String[2];
    function Add(a, b : Integer) : Integer; FAR;
    begin
      Add := a + b;
    end;
    function Mult(a, b : Integer) : Integer; FAR;
    begin
      Mult := a * b;
    end;
   procedure MakeTable(w, h:Integer; Op:TFunc; c : Char);
   var i, j : Integer;
       function Hext(i : Integer) : TSt2;
       var H : Char;
       begin
        H := HD[i div 16];
                    {выделили старшую цифру; можно i shr 4}
        if H = '0' then H := ' ';
                    {чтобы не печатать незначащий ноль    }
        Hext := H + HD[i mod 16];
                                              {или i and $F}
       end; {Hext}
  begin
   Write(c:2, ' I');
   for i := 1 to w - 1 do Write(HD[i]:4);
   Writeln;
   for i := 1 to w  do Write('----');
   Writeln;
   for i := 1 to h - 1 do            {  количество строк }
       begin
       Write(HD[I]:2, ' I');
       for j := 1 to w - 1 do
           Write(Hext(Op(i, j)):4);
       Writeln;
       end;
  end;                                {      MakeTable   }
begin
```

```
MakeTable(16, 16, Add, '+');
Writeln;
MakeTable(16, 16, Mult, '*');
end.
```

Переменные – процедуры и функции

Применение процедурного типа не ограничивается только одним описанием параметров – процедур или функций. Если есть тип, значит, могут быть и переменные такого типа.

Процедурные переменные по формату совместимы с переменными типа `Pointer` и после приведения типов могут обмениваться с ними значениями.

В предыдущей программе могут быть такие изменения. Добавим

```
var  Oper : TFunc;
     Cond : Char;
```

В главной программе выполним следующие действия:

```
Writeln('задай операцию (*/+)');
Readln(Cond);
Writeln;
case Cond of
     '*' : Oper := Mult;
     '+' : Oper := Add
     end;
MakeTable(16, 16, Oper, Cond); …
```

Замечание. Подпрограмма, которая присваивается процедурной переменной:

1) не должна быть стандартной процедурой или функцией.

Это ограничение можно обойти таким, например, приемом: вызов стандартной подпрограммы заключить в «оболочку».

```
var Func : function(x : Real) : Real;
  …
  function MySin(x : Real) : Real; FAR;
   begin
    MySin:=Sin(x)
   end;
  …
Func:=MySin;
```

2) не может быть вложена в другие подпрограммы;

3) не может быть подпрограммой специального вида, которая имеет спецификацию **`interrupt`** или конструкцию **`inline`**.

Переменные процедурных типов, чтобы быть совместимыми по присвоению значения должны иметь одинаковое количество формальных параметров и параметры на соответствующих позициях должны принадлежать одному и тому же типу (совпадать).

У результата функций должны совпадать типы значений, которые возвращаются.

Пример. Составить подпрограмму вычисления значения определенного интеграла по формуле трапеций.

$$J = \int_a^b f(x)dx \approx h\left(\frac{y_0 + y_n}{2} + \sum_{i=1}^{n-1} y_i\right), \quad \text{где} \quad h = \frac{b-a}{n}, \quad y_0 = f(a), \quad y_n = f(b),$$

$y_i = f(x_i)$, $x_i = x_{i-1} + h$, $i = 1, 2, \ldots, n-1$, $x_0 = a$, n – число отрезков разбиения интервала интегрирования.

По этой программе посчитать $I_1 = \int_{\pi/4}^{\pi/2} \frac{x^2 \ln x}{\sin x + x} dx$ и $I_2 = \int_{\pi/6}^{\pi/3} \frac{x^2}{\cos x + x} dx$ для некоторого заданного *n*.

Решение.

```
program Int;
const     n = 20;
type      TFunc = function(x : Real) : Real;
var
    fu: TFunc;

 function f1(x : Real) : Real; FAR;
  begin
   f1 := (x * x * ln(x)) / (Sin(x) + x)
  end;

 function f2(x : Real) : Real; FAR;
  begin
   f2 := x * x / (Cos(x) + x)
  end;

 function Trap( n:Integer; a,b:Real; f: TFunc) : Real;
 var
    h, s, x  : Real;
    i        : Integer;

 begin
  h := (b - a) / n;
  s := (f(a) + f(b)) * 0.5;
  x := a;
  for i := 1  to  n - 1  do
     begin
```

```
      x := x + h;
      s := s + f(x)
     end;
  Trap := h * s;
 end;

begin
 Writeln('n=', n);
 fu := f2;
 Writeln(Trap(n,pi/4,pi/2, f1),Trap(n,pi/6,pi/3,fu));
end.
```

2 СЕМЕСТР

1.13. Модули

Содержание

- Стандартные библиотечные модули.
- Модули пользователя.
- Подпрограммы в модулях.
- Комбинированный тип «запись».
- Обработка данных типа запись.

Стандартные библиотечные модули

Pascal дает программисту довольно широкий набор «встроенных» процедур и функций для реализации действий, которые наиболее часто встречаются при написании программ.

Различают процедуры и функции таких видов: арифметические, преобразования типов, управления строками на экране, управления памятью для динамических переменных, обработки строк, файлов, выполнения действий по выводу графических объектов и др.

Система программирования Turbo Pascal имеет модульную структуру, когда все стандартные средства выделены в отдельные группы, которые расположены в физически обособленных библиотеках – стандартных модулях. Эти библиотеки обеспечивают неограниченное расширение программных возможностей. Каждый модуль объединяет логически отделенную именованную группу типов данных, констант, переменных, процедур и функций.

Представление программы как единой языковой конструкции явилось преградой для эффективной организации коллективных разработок сложных систем. Подпрограммы, рассмотренные нами раньше (процедуры и функции), были частью программы.

Решающим шагом на пути преобразования языка Turbo Pascal в язык, подходящий для крупных разработок производственного и коммерческого назначения на современном уровне технологии программирования, явилось

введение понятия модуля. За счет введения модулей удалось ослабить ограничения на суммарный объем (≤ 64) кб готовых программ.

В языке Turbo Pascal модуль (unit) по определению считается отдельной программной единицей. Если подпрограмма является структурным элементом Pascal-программы и не может существовать вне нее, то модуль представляет собой отдельно хранимую и независимо компилируемую программную единицу.

В общем виде *модуль* – это совокупность (коллекция) программных ресурсов, предназначенных для использования другими модулями и программами. Под ресурсами будем понимать константы, типы, переменные, подпрограммы.

Turbo Pascal включает 10 стандартных модулей для реального режима DOS. В библиотеке TURBO.TPL содержатся модули SYSTEM, OVERLAY, DOS, CRT, PRINTER. Остальные модули (GRAPH, STRINGS, WINDOS, TURBO3, GRAPH3) располагаются в отдельных файлах с расширением TPU.

Программные ресурсы, сосредоточенные в стандартных модулях, образуют мощные пакеты системных средств, обеспечивающих высокую эффективность и широкий спектр применения системы Turbo Pascal.

Модуль SYSTEM подключается автоматически, является сердцем Turbo Pascal, поскольку ресурсы, которые он содержит, обеспечивают работу всех остальных модулей системы. Этот модуль содержит все стандартные функции, как математические (*exp, ln, sin*, ...), так и другие: поддерживает динамическое распределение памяти, целочисленную арифметику и с плавающей точкой, объединяющей подпрограммы ввода-вывода и др.

Crt – управляет дисплеем, клавиатурой и звуком.

DOS и WINDOS - обслуживают прерывания, выполняют проверку состояния дисков, содержат специальные средства обработки файлов, совершают управление операционным окружением, таймером (т. е. обеспечивают работу с функциями операционной системы MS DOS и Windows).

Graph – содержит огромный пакет графических средств.

Graph3 – поддерживает использование стандартных графических средств версии Turbo Pascal 3.0 (графические процедуры и функции этого модуля имеют другие названия по сравнению с процедурами и функциями модуля Graph).

Turbo3 – обеспечивает совместимость с версией Turbo Pascal 3.0.

Printer – обеспечивает быстрый доступ к устройству печати (устарел).

Overlay – содержит средства организации программ, которые по очереди совместно используют общую часть памяти.

Strings – дает возможность программе использовать строки, которые используются в Windows-приложениях.

Win – является приложением к модулю CRT, обеспечивая новые возможности при работе с окнами.

Модули пользователя

Модуль, кроме заголовка `Unit Unitname`, имеет три части – *интерфейсную*, *реализации* и *инициализации*.

В *интерфейсной части* модуля собраны описания объектов, которые доступны из других программ. Это видимые вне модуля объекты.

В *части реализации* содержатся рабочие объекты – скрытые, или невидимые.

В *части инициализации* описываются действия, которые будут выполнены при подключении модуля.

Общая структура модуля:

```
UNIT Unitname;
INTERFACE
```

Описание видимых объектов	Описание типов, констант, переменных, заголовков процедур, функций

```
IMPLEMENTATION
```

Описание скрытых объектов	Реализация процедур и функций, которые описаны в интерфейсной части, и вспомогательных алгоритмов, типов, констант и переменных

```
BEGIN
```

Операторы инициализации объектов модуля	Установка начальных значений переменных модуля перед его использованием

```
END.
```

При описании модуля используются служебные слова: `unit, interface, implementation`.

Пример 1.

```
unit Calendar;
interface
type
      Days        = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
      WorkingDays = Mon .. Fri;
      Months      = (Jan, Feb, Mar,  Apr, May, June,
                     Jule, Aug, Sept, Oct, Nov, Desem);
      Summer      = June .. Aug;
      Autumn      = Sept .. Nov;
      Spring      = Mar  .. May;
      Dayno       =    1 ..31;
      Yearno      = 1900 ..2020;
```

```
        Date        = record
                        Day   : Dayno;
                        Month : Months;
                        Year  : Yearno
                      end;
                {здесь можно описать подпрограммы
                 для получение дня недели
                 по заданному дню, месяцу, году...}

     implementation

    end.
```

Пример касается определений, связанных с датами и месяцами. Данный модуль, ввиду своей простоты, не содержит разделов реализации и инициализации.

Подключение модуля происходит в начале программы операторам `uses`.

Спецификация `uses` должна идти непосредственно после заголовка программы, так как она содержит определения данных и подпрограмм.

Если некоторый модуль подключается в модуль в раздел `interface` или `implementation`, тогда `uses` должен идти сразу после служебных слов `interface` или `implementation`.

Подпрограммы в модулях

Процедуры и функции могут использоваться в модулях наряду с другими Pascal-объектами. Считается, что заголовок подпрограммы является ее интерфейсом (видимый), а тело – реализацией (невидимо), так как заголовок содержит всю информацию, необходимую для вызова: ее имя, число и типы параметров, а для функций – и тип результата. А тело подпрограммы раскрывает алгоритм.

Значит, в интерфейсной части модуля представлены только заголовки процедур и функций, видимые (доступные) для других программ, а их полное описание, которое содержится в разделе реализации, может иметь сокращенный заголовок, состоящий только из служебного слова `procedure` или `function`, имя подпрограммы и «;». (Но можно идентично повторить и полный заголовок).

Пример 2. В следующем модуле собраны средства для работы с комплексными числами, которые дальше можно расширить, например, для:

а) на работу с массивами: перемножить матрицы, умножить матрицу на вектор и т. д.

б) на подсчет тригонометрических функций от комплексного аргумента и т. д.

```
unit CmplVals;
```

```
interface
type
      Complex = record
                  Re, Im : Real
                end;
    {заголовки процедур, которые реализуют
      операции над комплексными числами}
procedure  InitC  (R, I : Real;      var C : Complex);
procedure  AddC   (C1, C2 : Complex; var R : Complex);
procedure  MultC  (C1, C2 : Complex; var R : Complex);
procedure  DivC   (C1, C2 : Complex; var R : Complex);
procedure  WriteC (C : Complex);
function   ModC   (C : Complex) : Real;

implementation
                 {полное описание процедур}
procedure  InitC;
begin
 with C do
     begin
      Re := R;      Im := I
     end
end;
…   end.
```

Таким образом, механизм модулей позволяет спрятать детали реализации тех или иных программных подсистем.

Как результат, изменение реализации какой-нибудь подпрограммы при условии, что интерфейс модуля при этом останется неизменным, никак не отобьется на программах, его использующих.

Модуль компилируется таким же образом, как и программа, но поскольку модуль не является непосредственно выполняемой единицей, то в результате его компиляции образуется дисковый файл с расширением «.TPU» (Turbo Pascal Unit), при этом имя TPU-файла должно совпадать с именем файла с исходным текстом модуля.

Поэтому имя модуля не может состоять более чем из восьми символов.

И, если мы хотим создать модуль unit, то нужно, чтобы у файла, в котором он будет сохранен, имя было таким же, как и у модуля unit.

Рассмотрим программу, которая использует модуль CmpVals:

```
program UsingComplex;
uses CmplVals;
  {чтобы получить доступ к интерфейсным объектам модуля, необходимо указать в программе имя нужного TPU-файла}
var C1, C2, C3 : Complex;
    {заданный в модуле тип Complex, доступен здесь так же, как будто он определен в программе}
begin
 InitC(1,  2, C1);
 InitC(3,  4, C2);
 MultC(C1, C2, C3);
 WriteC(C3);
 DivC(C1,  C2, C3);
 WriteC(C3)
end.
```

В связи с использованием модулей возникают следующие важные моменты:

1. Может случиться, что идентификаторы интерфейсной части модуля частично пересекаются с идентификаторами программы. В этом случае идентификаторы программы «экранируют» (затемняют) одноименные идентификаторы модуля, например:

```
program Pr;
uses A, B; …
```

В таком случае идентификаторы программы `Pr` затемняют идентификаторы модуля `B`, а те затемняют идентификаторы модуля `A`.

2. Но если все же следует обратиться к одноименному данному из модуля, то следует образовать составное имя по принципу `Unitname.name`, структура которого похожа на селектор поля записи, например:

```
X := A.X;
```

Переменной `x` программы `Pr` присваивается значение переменной `x` модуля `A`.

3. Возможны случаи косвенного использования модулей.

```
unit A;
interface
  …
end.
unit B;
interface
uses A;
```

```
  …
end.
```

```
program P;
    uses B;  …
    end.
```

Достаточно указать только те модули, которые непосредственно используются в программе.

4. Схема использования модулей может образовывать древовидную структуру любой сложности, но при этом недопустим явное или косвенное обращение модуля к самому себе.

5. Если в модуле существует раздел инициализации, то операторы из этого раздела будут выполнены перед началом выполнения программы (или модуля), в которой используется данный модуль.

Отметим следующее:

1. Раздел **unit** содержит имя библиотечного модуля. Оно должно совпадать с именем дискового файла, где находится исходный текст модуля.

save *compile*

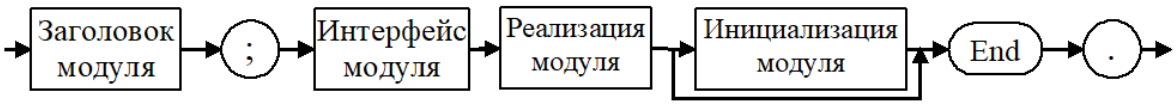
unit Stat ⇒ stat.pas (сохранили в файле) ⇒ stat.tpu

В *интерфейсной части* описываются константы, типы, переменные, процедуры и функции, которые являются глобальными, т. е. доступны основной программе (или модулю, который использует данный модуль).

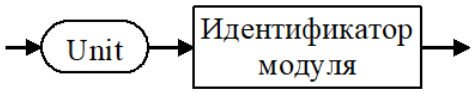
В *секции реализации* определяются модули всех глобальных подпрограмм. Кроме того, в ней описываются константы, переменные, подпрограммы, которые уже локальные.

2. *Секция инициализации* – последняя секция модуля. Начинается словом **begin**, и далее располагаются операторы инициализации, если они есть.

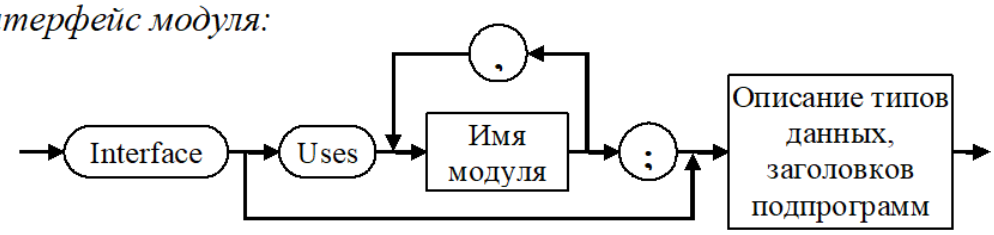
Структурная диаграмма модуля:



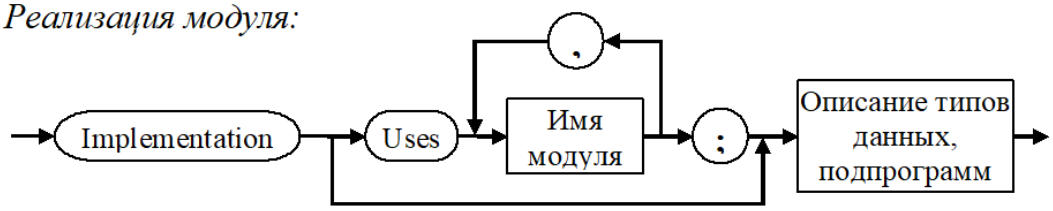
Заголовок модуля:



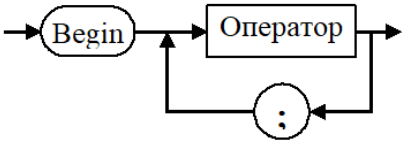
Интерфейс модуля:



Реализация модуля:



Инициализация модуля:



Встроенный компоновщик файлов ищет подключенный unit сначала в `TURBO.TPL` (`Turbo Pascal Library`), а потом в unit-директории, определенной в среде `Turbo`.

Свои модули можно помещать в `TURBO.TPL` специальной утилитой, но лучше построить свою `MyUnits` библиотеку.

Использование модуля:

- Открыть файл, содержащий текст программы, которая использует модуль `MyUnit.`
- В меню `Options` выбрать команду `Directories` и в диалоговом окне в поле `UnitDirectories` указать путь к модулю `MyUnit.`
- Далее через `Run` можно запустить на выполнение главную программу.

Модуль – это автономно компилируем программная единица.

В среде `TP` имеются средства, управляющие способом компиляции модулей. В меню `COMPILE` можно выбрать одну из следующих команд:

- `COMPILE`: при компиляции модуля или основной программы все заявленные модули должны быть предварительно откомпилированы и программе должны быть доступны все файлы *.`Tpu`. Путь к модулю задается в опции `Unit Directories` меню `OPTIONS/Directories`.
- `MAKE`: компилятор проверяет наличие `tpu`-файлов для каждого объявленного модуля. Если какой-либо из файлов не нашелся, система ищет одноименный файл с расширением `pas` (файл с исходным текстом) и его компилирует. Система перекомпилирует любой **`unit`**, который был модифицирован после последней компиляции. Если модуль не найден, то возникает ошибка компиляции.
- `BUILD`: существующие `tpu`-файлы игнорируются, система отыскивает и компилирует соответствующие `pas`-файлы для каждого объявленного модуля, подключаемого к данной программе. Программист должен обеспечить доступ к любому `pas`-файлу заявленного модуля **`unit`**.

Пример. В отдельном модуле собрать средства для работы с комплексными числами, которые объявляются как запись с двумя вещественными полями.

В главной программе отладить описанные процедуры и функции.

Данный модуль расширить:

а) на работу с одно и двумерными массивами (инициализация массива, распечатка массива, реализация задач линейной алгебры, например, перемножить матрицы, умножить матрицу на вектор и т. д.);

б) на подсчет тригонометрических функций от комплексного аргумента и т. д.

```
unit CmplVals;
interface
type
      Complex = record
                  Re, Im : Real
                end;
procedure  InitC  (R, I : Real;      var C : Complex);
procedure  AddC   (C1, C2 : Complex; var R : Complex);
procedure  MultC  (C1, C2 : Complex; var R : Complex);
procedure  DivC   (C1, C2 : Complex; var R : Complex);
procedure  WriteC (C : Complex);
function   ModC   (C : Complex) : Real;

implementation
                 {полное описание процедур}
procedure  InitC;
  begin
     with C do
       begin
         Re := R;      Im := I
       end
  end;
procedure  AddC (C1, C2 : Complex; var R : Complex);
  begin
    with R do
      begin
        Re := C1.Re+C2.Re;
        Im := C1.Im+C2.Im
```

```
     end
  end;
procedure  MultC (C1, C2 : Complex; var R : Complex);
  begin
    with R do
      begin
        Re := C1.Re*C2.Re– C1.Im*C2.Im;
        Im := C1.Re*C2.Im+C2.Re*C1.Im
      end
  end;
function  ModC (C : Complex) : Real;
  begin
    ModC := C.Re*C.Re+C.Im*C.Im;
  end;
procedure  DivC (C1, C2 : Complex; var R : Complex);
  var Tmp : Real;
  begin
T    mp := ModC(C2);
     with R do
       begin
         Re := (C1.Re*C2.Re+C1.Im*C2.Im)/Tmp;
         Im := (C2.Re*C1.Im–C1.Re*C2.Im)/Tmp
       end
  end;
procedure  WriteC (C : Complex);
  begin
    with C do
      begin
        Write(Re); if Im=0 then Exit;
        if Im > 0 then Write ('+');
        Write(Im); Write ('i');
      end
  end;
end.
```

1.14. Файлы в языке Pascal

Содержание

- Файловые типы.
- Операции над файлами:
  - установочные и завершающие операции,
  - специальные операции.
- Операции ввода-вывода для файлов с типом.
- Последовательный и прямой доступ к файлу с типом.

Файловые типы

Представители файла в Pascal-программе – переменные файловых типов. Для описания файловой переменной используются слова **`text`**, **`file`** или словосочетание **`file of`**. Например, описание переменной

```
var   f : file of Integer;
```

понимается как определение под именем `f` последовательности, расположенной на некотором внешнем запоминающем устройстве (например, на диске) и состоящей из неясного количества целых чисел, которые называются *компонентами* или *элементами файла*.

При работе с файлом с каждой переменной файлового типа согласуется «текущий указатель» файла, который можно понимать как скрытую переменную (т. е. не очевидно описанную вместе с файловой переменной). Текущий указатель обозначает («указывает» на) некоторый конкретный элемент файла.

Как правило, все действия с файлом (чтение из файла, запись в файл) выполняются поэлементно, причем в этих действиях участвует тот элемент файла, на который указывает текущий указатель.

На практике ввод-вывод выполняется целым блоком элементов через некоторый системный буфер ввода-вывода.

Будем понимать, что текущий указатель – это окно, которое перемещается по файлу и через которое мы «видим» элементы файла, доступные для обработки.

Если файловый тип задается в программе при помощи служебных слов **`file of`**, за которыми идет тип элементов файла (базовый тип), то базовый тип может быть любым типом, за исключением файлового и типа «объект» (который будет рассмотрен позже). Кроме того, в качестве базового типа не допускается запись, одним из полей которой является файл или объект.

Операции над файлами

В отличие от переменных других типов язык Pascal не содержит встроенных операций над собственно файловыми переменными. Операции с файлами реализованы в виде стандартных подпрограмм, собранных в модуле `System`, который присоединяется автоматически.

Операции с файловыми переменными можно разбить на четыре основные группы:

- установочные и завершающие операции;
- специальные операции;
- собственно ввод-вывод;
- перемещение по файлу.

Установочные и завершающие операции

В эту группу входят следующие процедуры.

1. `Assign(f, Name)` связывает файл `Name` с файловой переменной `f`.

2. `Reset(f)` открывает доступ к существующему файлу `f`.

3. `Rewrite(f)` создает и открывает новый файл `f`.

4. `Flush(f)` переписывает данные из буфера в файл.

5. `Close(f)` закрывает файл.

Здесь `f` – переменная файлового типа.

Рассмотрим процедуры подробнее.

1. Процедура `Assign(f, Name)` имеет первый параметр – имя файловой переменной f, второй параметр – строковое выражение – полное имя файла. Процедура предназначена для установления связи между конкретным файлом (набором данных) на внешнем носителе и переменной файлового типа. Например:

```
Assign(f, 'd:\mydir\myfile.dat');
```

После выполнения этого оператора предполагается, что файловая переменная `f` будет связана с дисковым файлом `myfile.dat`, расположенном в каталоге `mydir` корневого каталога диска `d`. (Проверка на корректность не делается.) Эта процедура всегда предшествует другим процедурам работы с файлами.

Вместо набора данных (файла) может быть любое устройство ввода-вывода: клавиатура, печатающее устройство или дисплей. Их еще называют псевдофайлами MS DOS.

2-3. Процедуры `Reset (f)` и `Rewrite (f)` предназначены для открытия файла, где `f` — файловая переменная, назначенная предварительно оператором `Assign` файлу. Под открытием в данном случае понимается поиск файла на внешнем носителе; создание специальных системных буферов для обмена с ним (операционная система ставит в соответствие каждому открываемому файлу скрытый от нас обработчик файлов со своим номером); установка текущего указателя файла на его начало. Все элементы файла с типом условно нумеруются: 0, 1, 2, ...

Процедура `Reset(f)` предполагает, что файл, который открывается, уже существует, в противном случае возникает ошибка.

Процедура `Rewrite(f)` допускает, что файл, который открывается, может не существовать, тогда она создает заданный файл. Если же файл существует, тогда `Rewrite` очищает его.

В обоих случаях, если файл f был открыт, он предварительно закрывается. Но связь с существующим файлом не нарушается (она настроена через Assign).

4. Процедура Flush(f) завершает обмен с файлом без его закрытия.

Обмены с файлами всегда реализуются через некоторый буфер в оперативной памяти, поэтому в процессе записи в файл последние элементы, которые записываются, могут еще «остаться» в буфере. Процедура Flush вызывает принудительное сбрасывание этих элементов в файл.

5. Процедура Close (f) завершает действие с файлом f:

а) переписывает информацию из буфера в файл (если файл для вывода);

б) устраняет внутренние буферы, созданные при открытии этого файла;

в) устраняет связь файла с файловой переменной f (освобождает обработчик файла для других работ).

После этого файловую переменную можно связать с помощью Assign с каким-либо другим файлом или устройством.

Заметим, что при окончании работы программы происходит автоматическое закрытие всех открытых в программе файлов. Однако хорошим правилом является явное закрытие файлов после окончания работы с ними, ибо может потеряться часть информации, которая осталась в буфере.

6. Процедура Append(f) – открывает существующий тестовый файл f для добавления.

Если файл отсутствует на диске, то возникает ошибка ввода-вывода.

Специальные операции

Эта группа операций предназначена для действий с элементами файловой системы MS DOS – каталогами и именами файлов, позволяя создавать и ликвидировать файлы, работать с атрибутами файлов и так далее.

Подробнее об этих операциях можно прочесть в фирменных руководствах по языку. Рассмотрим только 2 следующие:

- Erase(f) уничтожение файла на диске, который был связан с файловой переменной f. Если файл не существует, то возникает ошибка ввода-вывода;
- Rename(f, name) – переименование файла.

Эти процедуры работают с неоткрытыми файлами: требуется выполнить только процедуру Assign, но не выполнять Reset или Rewrite.

Rename(f, newname) переименовывает неоткрытый файл f. Новое имя задается строкой newname. При этом нельзя изменять имя диска и путь к файлу, изменяется только собственное имя физического файла.

Операции ввода-вывода для файлов с типом

В эту группу входят две операции (процедуры) Read и Write, которые реализуют чтение информации из файла и запись информации в файл соответственно.

Обмен данными происходит через буфер ввода-вывода, размер которого устанавливается автоматически, исходя из размера элементов файла.

Как бы файл с типом ни был открыт (Reset или Rewrite), в модуле System описана переменная FileMode, значение которой становится равной 2 (значение 0 означает только чтение, значение 1 – только запись). Оно показывает, что из файла можно и читать, и в файл можно записывать данные.

В отличие от многих других процедур, Read и Write могут вызываться с разным числом параметров. Формат операторов:

Read(f, v1, ..., vn) и

Write(f, v1, ..., vn).

Список v1, ..., vn может состоять из нескольких или из одной переменной базового типа файла.

Процедура Read предназначена для чтения значений из файла. Первый параметр – имя файловой переменной, к которой была применена одна из операций открытия (Reset или Rewrite). Далее должны идти переменные, в которых будут помещены значения из файла. Тип этих переменных должен совпадать с базовым типом файла.

Выполнение процедуры Read происходит следующим образом. Начиная с текущей позиции указателя файла, последовательно будут читаться значения, находящиеся в файле, и присваиваться очередной переменной из тех, которые перечислены в списке ввода. После каждого действия по чтению данного из файла, указатель файла будет перемещаться на следующий элемент. Если в процессе выполнения процедуры Read текущий указатель файла будет установлен на позицию последнего элемента файла, и он будет прочитан, тогда возникнет ситуация «*конец файла*».

Возникновение ситуации «конец файла» можно проверить при помощи встроенной логической функции EoF(F) с параметром F – файловой переменной. Функция EoF(F) возвращает логическое значение true, если достигнут конец файла, и false – в противном случае. При обращении к файлу для чтения, у которого EoF имеет значение true, система дает фатальную ошибку 100 (считывание после конца файла).

При записи в файл истинность функции EoF (f) означает, что очередная операция записи поместит информацию в конец данного файла.

Процедура Write позволяет записывать информацию в файл. Первым параметром этой процедуры должна быть файловая переменная, открытая процедурой Reset или Rewrite. Далее должен идти список переменных, тип которых совпадает с базовым типом файловой переменной. При записи в файл записывается внутреннее представление очередного элемента. В файле элементы занимают один и тот же объем памяти в соответствии со своим типом.

Последовательный и прямой доступ к файлу с типом

Данная группа операций позволяет произвольно изменять последовательность выполнения операций чтения и записи. Сюда входят следующие процедуры.

- Seek(f, N) устанавливает указатель на элемент с номером N, где N имеет тип Longint.

- `Truncate(f)` уничтожает все элементы файла, начиная с места текущего указателя.

Можно пользоваться и двумя дополнительными функциями:

- `FileSize(f)` возвращает размер файла (количество элементов);
- `FilePos(f)` возвращает номер элемента, на который установлен текущий указатель.

Процедура `Seek` позволяет явно изменить значение текущего указателя, установив его на элемент файла с заданным номером (нумерация начинается с нуля). Это фактически прямой доступ к элементу с заданным номером. После выполнения процедуры `Seek` дальнейшие операции чтения или записи будут проводиться, начиная с установленной позиции указателя.

Примеры.

`Seek(f, 0)` → установка указателя на начало файла.

`Seek(f, FilePos(f)+1)` → пропуск одного элемента.

`Seek(f, FileSize(F))` → установка текущего указателя непосредственно за последним элементом файла.

Рассмотрим типовые алгоритмы для работы с файлами.

1. Общая структура фрагмента программы, предназначенной для чтения из файла с целью последующей обработки данных:

```
Assign(f, '…');
Reset(f);
while not eof(f) do
   begin
    Read(f, a);
    …
   end;
Close(f);
```

2. Общая структура фрагмента программы, предназначенной для записи в файл:

```
Assign(f, '…');
Rewrite(f);
 {Организовать цикл на количество записываемых элементов}
    begin
       {Получение данного для записи}
       Write(f, a);
    end;
Close(f);
```

3. Общая структура фрагмента программы, предназначенной для добавления в файл:

```
Reset(f);
Seek(f, FileSize(f));
 {Организовать цикл на количество записываемых элементов}
```

```
      begin
        {Получение данного для записи}
        Write(f, a);
      end;
 Close(f);
```

Некоторые алгоритмы работы с типизированными файлами

Задание. Прочитав информацию из текстового файла, создать типизированный файл для следующей задачи.

Задача. В типизированном файле существует информация о студентах: фамилия, имя, отчество (30 символов), возраст (2 цифры). Обновить данные, увеличив возраст всех студентов на единицу.

Алгоритм. Нецелесообразной будем считать следующую последовательность действий: читаем очередную запись, корректируем возраст, оператором `Seek(f, FilePos(f)–1)` перемещаемся на один элемент назад и записываем откорректированную запись. Ведь может случиться так, что возникнет непредвиденное аварийное завершение программы, и наш файл будет фактически испорчен. Поэтому на основе исходного файла создадим новый файл с откорректированной информацией. Если все хорошо закончится, уничтожим исходный файл, и новый переименуют на исходный файл.

```
program Correct;
type
     Zap = record
             FIO : String[30];
             Age : 0..99;
           end;
var
      f, fRes : file of Zap;
      Z       : Zap;
begin
 Assign(f,    'C:\1_курс\Baza.dat' );
 Assign(fRes, 'C:\1_курс\Baza1.dat');
 Reset(f);   Rewrite(fRes);
 while not eof(f) do
      begin
       Read(f, Z);
       Inc(Z.Age);
       Write(fRes, Z);
      end;
 Close(f);        Close(fRes);
 Writeln('Ok');   Readln;
 Assign(f, 'C:\1_курс\Baza.dat');
 Erase(f);
 Writeln('Ok');   Readln;
 Assign(f, 'C:\1_курс\Baza1.dat');
```

```
 Rename(f, 'C:\1_курс\Baza.dat');
 Close(f);
 Writeln('Ok');
 Readln;
end.
```

Обработка ошибок ввода-вывода

Компилятор языка Pascal позволяет генерировать исполняемый код в двух режимах: с проверкой корректности ввода-вывода и без нее. По умолчанию включена директива компиляции режима проверки {$I +} ({$I-} – режим проверки отключен).

При включенном режиме проверки любая ошибка ввода-вывода будет фатальной: программа прервется и выдаст номер ошибки. Возможные номера ошибок находятся в диапазоне от 2 до 200. Расшифровка кодов приводится в специальных таблицах и в строке – состояния завершения программы.

Если отключить режим проверки, тогда при возникновении ошибки программа уже не будет прерываться, а продолжит работу со следующего оператора. При этом код ошибки сохранится в предопределенной системной переменной `InOutRes`, но результат операции ввода-вывода, которая вызвала ошибку, будет неопределенным.

Однако при опросе кода завершения операции обмена лучше пользоваться специальной функцией Pascal `IOResult`. При успешном выполнении операций ввода-вывода обращение к `IOResult` дает 0, а ненулевое результат свидетельствует об ошибке.

Если программист захочет предусмотреть личную реакцию на ошибочные ситуации, тогда можно опросить функцию `IOResult`, которая вернет значение типа `Integer` – код (статус) последней выполненной операции ввода-вывода. Вызов функции `IOResult` очищает внутренний флаг ошибок (т. е. сбрасывает его в 0), и поэтому еще один вызов функции к одной и той же операции ввода-вывода даст некорректный результат.

Использование этой функции возможно только тогда, когда отключена стандартная проверка операций ввода-вывода: {$I-}. Рассмотрим следующий пример.

```
var
     Code : Integer; f : file;
      …
Assign(f, 'd:\myfile. dat');
{$I–}              {отключаем автоматический контроль,
                    иначе система остановит работу
                    при возникновении ошибки}
Reset(f);          {открыть существующий файл}
Code := IOResult;  {получили код результатат предыдущей
                    операции}
{$I+}              {включаем автоматический контроль}
if Code<>0 then
  begin
```

```
        Write('ошибка при открытии файла: ');
        case  Code   of
           2 : Write('файл не найден ');
           3 : Write('путь не найден ');
           4 : Write('слишком много открытых файлов');
           5 : Write('доступ к файлу запрещен');
           6 : Write('испорчена файловая переменная');
          12 : Write('некорректный код доступа к файлу');
         102 : Write('файлу не дано имя')
           else
        end; {case}
         Halt;
    end;
```

Приведенный выше фрагмент кода можно использовать при написании специальной функции для анализа любой операции ввода-вывода, и в дальнейшем операции ввода-вывода проводить с обязательной диагностикой ошибок. Рассмотрим фрагмент программы проверки наличия файла с данными.

Пример.

```
Assign(f, 'NoFile.dat');
{$I–}   Reset(f);                {попытка открыть файл}
{$I+}
if IOResult <> 0 then
   Writeln('файл не найден или не читается')
   else
       begin
        Read(f, …);              {можно работать}
           …
        Close(f)
       end;
```

Аналогично можно построить функцию анализа существования файла:

```
function  FileExists
     (FName : String; var Cod : Integer) : Boolean;
var
      f : file;                 {тип файла не существенен}
begin
 Assign(f, FName);
 {$I–}
  Reset(f);
 {$I+}
 Cod := IOResult;
 if Cod = 0 then
   begin
    FileExists := true;
    Close(f);
   end
 else  FileExists := false
```

```
end;
```

Слияние двух отсортированных последовательностей

Тривиальный алгоритм слияния, когда в конец одной последовательности дописывается вторая, а потом происходит ее сортировка, отбросим как нецелесообразный, так как при таком подходе требуется много неоправданных затрат на сортировку. Рассмотрим следующую задачу.

Задача. Пусть некий типизированный файл '`f1.dat`' содержит первую последовательность элементов, отсортированных по возрастанию, а '`f2.dat`' – другую. Получить '`f.dat`' – объединенную последовательность с ненарушенным порядком возрастания, причем файлы разрешается читать только один раз.

Интуитивный алгоритм

Пусть

A={2, 4, 6, 8, 17, 20, 22, 30, 80, 82, 84, 86, 90}
B={1, 3, 4, 9, 15, 28}

Будем читать по одному элементу из каждой последовательности: a=2; b=1. Меньший из них пишется в C, и если это a, тогда читается новый элемент из A в a; иначе – из B в b. Так делаем до тех пор, пока одна из двух последовательностей не иссякнет.

На нашем примере первой закончится последовательность B. Тогда элементы другой последовательности, которые остались, просто дописываем в C.

Первый этап:

C_1={1, 2, 3, 4, 4, 6, 8, 9, 15, 17, 20, 22, 28, 30}

Второй этап:

C=C_1∪{80, 82, 84, 86, 90}

Значит, программа должна анализировать, будет ли второй этап и какую последовательность еще нужно добавить к C (из A или из B).

Здесь сталкиваемся с проблемой, являющейся результатом специфики работы функции `Eof` в Turbo Pascal. Логическая функция `Eof(f)` возвращает значение `True` при чтении последнего элемента файла. Поэтому последний прочтенный элемент не будет анализироваться в цикле и придется этот анализ выполнять после выхода из цикла.

Задание. Этот алгоритм запрограммируйте самостоятельно как на примере файлов, так и на примере массивов.

Первое улучшение алгоритма

Есть первое улучшение этого алгоритма – добавление фиктивного элемента в каждую последовательность, а именно:

A′=A∪{$b_{\text{последний}}$}

B′=B∪{$a_{\text{последний}}$}

Поскольку $b_{\text{последний}}$ и $a_{\text{последний}}$ – наибольшие элементы своих последовательностей, то ситуация улучшится и нужно будет записать только

элементы из исходной последовательности, которые остались необработанными. Получим следующую программу.

Программа 1.

```
program Var2;
type
     fil = file of Integer;
var
     f1, f2, f : fil;

procedure Sozd(nam1, nam2 : String);
 var
         f : fil;
         t : text;
         a : Integer;
 begin
  Assign(t, nam1);
  Assign(f, nam2);
  Reset(t);
  Rewrite(f);
  while not eof(t) do
      begin
       Read(t, a);
       Write(f, a);
      end;
  Close(f);
  Close(t);
 end;
 procedure Show(var f : fil; st : String);
  var
      a : Integer;
  begin
   Writeln(st);
   Seek(f, 0);
   while not eof(f) do
        begin
        Read(f, a);
        Write(a : 4);
        end;
  Writeln;
  end;
  procedure Union(var f1, f2, f : fil);
   var
       a, b : Integer;
   procedure Astatok(var f3, f : fil; a : Integer);
     begin
     Write(f, a);
```

```
     while not eof(f3) do
          begin
           Read(f3, a);
           Write(f, a);
          end;
     end;
   begin
    Seek(f1, filesize(f1)-1);
    Read(f1, a);
    Seek(f2, filesize(f2)-1);
    Read(f2, b);
    Write(f1, b);
    Write(f2, a);
    Seek(f1, 0);
    Read(f1, a);
    Seek(f2, 0);
    Read(f2, b);
    while not eof(f1) and not eof(f2) do
        begin
         if a>b then
          begin
            Write(f,  b);
            Read(f2, b)
          end
         else
          begin
            Write(f,  a);
            Read(f1, a)
          end;
        end;
   if eof(f1) then Astatok(f2, f, b)
              else Astatok(f1, f, a);
   Seek(f, Filesize (f)-1);
   Truncate(f);
   Seek(f1, Filesize (f1)-1);
   Truncate(f1);
   Seek(f2, Filesize (f2)-1);
   Truncate(f2);
   end;
begin
Sozd('f1.txt', 'f1.dat');
Sozd('f2.txt', 'f2.dat');
Assign(f1, 'f1.dat');
Assign(f2, 'f2.dat');
Assign(f,  'f.dat');
Reset(f1);
Reset(f2);
```

```
Rewrite(f);
Show(f1, '*****  1 file  ********');
Show(f2, '*****  2 file  ********');
Union(f1, f2, f);
Show(f,  '*****  3 file  ********');
Close(f1);
Close(f2);
Close(f3)
end.
```

Второе улучшение алгоритма

Есть еще одно улучшение предыдущего алгоритма – добавление двух фиктивных элементов в каждую последовательность, а именно:

$A' = A \cup \{b_{последний}\} \cup \{b_{последний}\}$

$B' = B \cup \{a_{последний}\} \cup \{a_{последний}\}$

Поскольку $b_{последний}$ и $a_{последний}$ есть наибольшие элементы своих последовательностей, то положение улучшится, и не надо будет анализировать ситуацию, в которой из исходных последовательностей остались необработанные элементы.

Рассмотрим это улучшение на следующих данных.

Первый файл:

1 3 4 9

Второй файл:

04 6 8 10 12 14 16 22

Результат:

01 3 4 4 6 8 9 10 12 14 16 22

Программа 2.

```
program Var3;
type
     Fil = file of Integer;
var
     f1, f2, f : fil;
procedure Sozd(nam1, nam2 : String);
var
         f : fil;
         t : text;
         a : Integer;
begin
 Assign(t, nam1);
 Assign(f, nam2);
 Reset(t);
 Rewrite(f);
 while not eof(t) do
      begin
       Read(t, a);
```

```
      Write(f, a);
     end;
 Close(f);
 Close(t);
end;
 procedure Druk(var f : fil; st : String);
  var
      a : Integer;
  begin
  Writeln(st);
  Seek(f, 0);
  while not eof(f) do
       begin
        Read(f, a);
        Write(a : 4);
       end;
  Writeln;
  end;
 procedure Union(var f1, f2, f : fil);
 var
     a, b : Integer;
 begin
  Seek(f1, filesize(f1)-1);
  Read(f1, a);
  Seek(f2, filesize(f2)-1);
  Read(f2, b);
  Write(f1, b, b);
  Write(f2, a, a);
  Seek(f1, 0);
  Seek(f2, 0);
  Read(f1, a);
  Read(f2, b);
  while not eof(f1) and not eof(f2) do
      begin
       if a=b then
         begin
          Write(f, a, b);
          Read(f2, b);
          Read(f1, a)
         end
       else
        if a>b then
         begin
          Write(f, b);
          Read(f2, b)
         end
       else
```

```
           begin
            Write(f, a);
            Read(f1, a)
           end;
         end;
         if a = b then Seek(f, filesize(f)-2)
                  else Seek(f, filesize(f)-1);
         Truncate(f);
         Seek(f1, filesize(f1)-2);
         Truncate(f1);
         Seek(f2, filesize(f2)-2);
         Truncate(f2);
 end;
begin
 sozd('f2.txt', 'f1.dat');
 sozd('f1.txt', 'f2.dat');
 Assign(f1, 'f1.dat');
 Reset(f1);
 Assign(f2, 'f2.dat');
 Reset(f2);
 Assign(f, 'f.dat');
 Rewrite(f);
 Druk(f1, '*****  1 file  ********');
 Druk(f2, '*****  2 file  ********');
 Union(f1, f2, f);
 Druk(f,  '*****  3 file  ********');
 Close(f);
 Close(f1);
 Close(f2)
end.
```

Текстовые файлы

При записи в текстовый файл данных (чисел, строк, логических значений и т. д.) они превращаются в символьный вид.

При чтении из текстового файла данные автоматически преобразуются из текстового представления во внутреннее.

Текстовые файлы можно обрабатывать только последовательно.

Ввод и вывод нельзя выполнять одновременно для одного и того же текстового файла.

Представителем текстового файла в программе является переменная файлового типа, которая должна быть описана так:

```
var
    Texinf : text;
```

Чтобы наглядно показать различие в представлении информации для двух видов файлов – текстовых и файлов с типом, обратимся к записям с типом `String[11]`.

Представление строк в текстовом файле следующее (таблица 23):

Таблица 23 – Представление строк в текстовом файле.

Первая строка файла:	`МАМА#13#10`
Вторая строка файла:	`ПОШЛА#13#10`
Третья строка файла:	`В МАГАЗИН#13#10`

Общая длина фразы для текстового файла будет 25 байт (6 байт – первая строка, 7 – вторая, 11 – третья, 1 – символ конца файла).

Представление строк в типизированных файле с типом `String[11]` (таблица 24):

Таблица 24 – Представление строк в типизированном файле.

Первая запись:	`#4МАМА#0#0#0#0#0#0`
Вторая запись:	`#5ПОШЛА#0#0#0#0#0#0`
Третья запись:	`#9В МАГАЗИН#0#0`

Здесь на месте символа #0 может стоять любой другой.

Общая длина фразы для типизированных файлов будет в 36 байт.

Для выполнения работ над строками в режиме записи или чтения предпочтение отдается текстовым файлам. Чтобы создать текстовый файл, лучше всего пользоваться любым текстовым редактором.

Когда вы разрабатываете программу, которая требует ввода трех и более чисел, то рекомендуется создать вспомогательные текстовые файлы с тестовыми вариантами наборов данных, и не тратить время на ввод данных с клавиатуры на каждый прогон программы. Но, конечно же, возможны и другие способы.

Процедуры для работы с текстовыми файлами

Процедуры `Assign`, `Close`, `Flush`, функция `EoF` одинаковы для типизированных и текстовых файлов.

`Append(F)` – открывает существующий файл `F` для добавления.

`Rewrite(F)` – создает новый текстовый файл, к которому можно только добавлять строки. Если файл с таким именем уже существует на диске, то он уничтожается и создается новый текстовый файл.

`Reset(F)` – можно применять только к существующему файлу, после чего из этого файла можно читать только последовательно.

Если новый текстовый файл закрывается, к нему автоматически добавляется маркер конца файла.

Второе обращение к `Reset(F)` установит текущий указатель снова в начало файла `F`.

Чтение информации с файлов типа `Text` нами было подробно рассмотрено, когда рассматривался стандартный файл `Input`.

`Read([F,] V1[,V2,…,Vn])` – читает информацию из файла `F` (когда `F` присутствует, иначе – с клавиатуры) в заданные переменные.

Замечание. Здесь и далее в квадратных скобках пишется информация, которая может как присутствовать – тогда скобки надо опустить, так и отсутствовать вместе со скобками.

Процедура чтения зависит от типа текущей переменной.

В символьную переменную читается очередной символ из файла, включая символы-разделители (`CR ≡ #13`, `LF ≡ #10`, `^Z ≡ #26`).

При чтении в переменную арифметического типа пропускаются пробелы и символы табуляции, и считывается значение арифметической константы до появления пробела, символа табуляции, маркера конца строки или файла. (Символ-ограничитель не считывается.)

При чтении данных в строковую переменную передается или столько символов, какова ее длина в объявлении переменной, или до появления маркера конца строки или файла. (Маркер в строку не заносится.)

`Readln` – отличается от `Read` тем, что, после считывания данных в перечисленные переменные, пропускаются все символы, которые остались в данной строке, в том числе и маркер конца строки, и происходит переход на новую строку.

`Write([F,] V1 [,V2,…,Vn])` – записывает информацию в файл `F` (если он задан, иначе – на экран) из заданных переменных или выражений. Процедура записи зависит от типа данных в списке вывода. Для символьных, арифметических и строковых данных выводится их значение, а для `Boolean` – строка **`true`** или **`false`**. Используется и форматированный вывод.

`Writeln` – отличается от `Write` тем, что после вывода в файл всех данных записывается маркер конца строки.

Функции для работы с текстовыми файлами

`Eof[(F)] : Boolean` – возвращает `true`, если следующим за последним прочитанным символом является маркер конца файла `F`.

Если файловая переменная `F` отсутствует, тогда подразумевается стандартный файл ввода.

`Eoln[(F)] : Boolean` – возвращает `true`, если следующим за последним прочитанным символом является маркер конца строки (или `Eof`).

При отсутствии аргумента подразумевается стандартный файл ввода.

Если `Eof(F)` – `true`, то и `Eoln(F)` – `true`.

`SeekEoln[(F)] : Boolean` проводит поиск конца текущей строки.

Эта функция пропускает все символы-разделители (пробелы и символы табуляции) в строке и устанавливает текущий указатель файла `F` или на конце строки и тогда функция возвращает `true`, или на первом значащем символе и тогда функция возвращает `false`.

`SeekEof[(F)] : Boolean` возвращает `true`, когда указатель файла `F` находится на маркере конца файла.

Функция пропускает все символы-разделители и также символы концов строк (т. е. переходит со строки на строку в поисках или конца файла, или первого значащего символа).

Практика работы с текстовыми файлами

Далее рассмотрим некоторые фрагменты из программ.

Задача. Написать процедуру, которая выбирает режим добавления в существующий текстовый файл или создания файла, если набор данных не нашелся при открытии.

```
procedure OpenWrite(var  f: text; FileName : String);
 begin
  Assign(f, FileName);
  {$I-}
  Append(f);
  {$I+}              {попытка открыть файл для добавления}
  if IOResult <> 0 then
    Rewrite(f);
       {если файл не может быть открыт, тогда создать его}
  end;
```

Файлы без типа

Файл без типа состоит из элементов одинакового размера, структура которых не имеет значения или неизвестна. Файлы без типа применяются для высокоскоростных программ обмена между оперативной и внешней памятью. Нетипизированный файл объявляется так:

```
var    f : file;
```

Для работы с такими файлами используют следующие процедуры и функции:

`Assign, Rewrite, Reset, Blockread, Blockwrite, Seek, Eof, FileSize, FilePos.`

Обмен с типизированным файлам осуществляется так: вводится-выводится значение данного в форме его внутреннего представления. Вспомним, что при обмене через текстовый файл идет преобразование последовательности символов во внутреннее представление данного при вводе и наоборот – при выводе.

Нетипизированный файл рассматривается в языке Pascal как совокупность байт-блоков.

Любой файл, который был подготовлен как файл `text` или файл с типом, можно открыть и начать работу с ним, как с нетипизированным набором данных.

Для таких файлов не нужно тратить время на преобразование данных или поиск управляющих последовательностей. Достаточно лишь считать содержимое файла в определенную память.

Для нетипизированных файлов самым важным параметром является длина блока в байтах, и этот параметр или присутствует в процедурах открытия

файла, чтения, записи, или берется по умолчанию. Однако суммарный объем разового обмена не должен превышать 64 кб.

Рассмотрим подробнее перечисленные выше процедуры:

`Rewrite(F [, Size])` – создает новый файл `F`: если файл существует, то он уничтожается и создается новый. Второй параметр `Size` определяет размер блока в файле; если `Size` отсутствует, то считается размер блока в 128 байт.

`Reset(F [, Size])` – открывает существующий файл `F`. В файл можно записывать информацию и читать из него. Второй параметр – как в предыдущей процедуре.

`Blockread(f, Buf, count [,Result])` – читает из файла `F` `count`-записей (блоков) в буфер ввода `Buf`. Если есть четвертый параметр `Result`, тогда он получает значение реального числа прочитанных записей (блоков). Если читается последняя порция count-записей (блоков) из файла, тогда она может быть либо пустой, либо неполной. Если же четвертый параметр отсутствует и количество востребованных (`count`) и реально прочитанных записей (блоков) не совпадает, тогда возникает ошибка ввода-вывода.

Если четвертый параметр присутствует, тогда ненормальное завершение ввода-вывода фиксируется четвертым параметром.

`Blockwrite(f, Buf, count [, Result])` – записывает в файл `count`-записей из буфера ввода `Buf`. Если указан четвертый параметр `Result`, тогда он получает реальное число записанных записей (блоков). Если, например, на диске не хватает места, тогда количество записанных элементов может не совпадать с заданным в `count`.

Если четвертый параметр не указан и количество требуемых и реально выведенных записей не совпадает, тогда возникает ошибка ввода-вывода.

Пример. Рассмотрим использование файлов без типа для ускорения обмена данных между внешней и оперативной памятью.

```
program Pr_Best;
var
     A, B, C : array[1..2000] of Real;
                    {компилятор распределяет их в цепочку
                     один за другим (специфика Pascal)}
     f       : file;
     Rez     : Integer;
begin
      ...                {заполняем массивы}
 Assign(f, 'ABC.dat');
 Rewrite(f, Sizeof(A));
 Blockwrite(f, A, 3, Rez);
                         {или три раза: из A, из B, из C}
 if Rez = 3 then Writeln('Okey')
            else Writeln('Error');
 Close(f); …
 end.
```

Если нужно быстро сбросить информацию из файла в память, тогда возможно следующее использование файла без типа:

```
… Reset(f, Sizeof(A));
Blockread(f, A, 3, Rez);
if Rez = 3 then Writeln('Okey')
           else Writeln('Error');
                        {дальше с массивами A, B, C
                         можно работать по отдельности}
Close(f);…
```

Сравните описанный здесь вариант обмена данных между внешней и оперативной памятью с обменом при помощи типизированных файлов.

1.15. Специальные средства Turbo Pascal. Модуль System

Содержание

- Указатели и динамические структуры данных.
  - создание динамических переменных,
  - операции,
  - доступ к переменной по указателю.
- Действия над динамическими переменными.
- Нетипизированные указатели.

Указатели и динамические структуры данных

Создание динамических переменных

При выполнении программы каждая объявленная в ней переменная получает свой адрес в оперативной памяти. Программисту не нужно заботиться о механизме определения адресов, это делается автоматически при трансляции.

В языке Pascal существуют два способа распределения памяти для переменных: статический и динамический.

Всем переменным, объявленным в программе или в интерфейсной части модуля, выделяются в определенном месте оперативной памяти (сегменте данных) фиксированные участки оперативной памяти в соответствии с их типом. Это статическое распределение памяти.

При динамическом распределении памяти есть возможность создавать новые, не объявленные заранее переменные и размещать их на свободные участки в динамической области оперативной памяти. Это достигается путем использования указателей.

Указатель – это элемент данных, представляющий собой ссылку (адрес) на начало определенного блока оперативной памяти (ячейку), где может содержаться значение данных заявленного типа. Сам указатель занимает 4 байта.

В IBM-совместимых компьютерах память условно разделена на так называемые *сегменты*. Адрес каждого байта составляется из адреса сегмента и адреса смещения относительно начала сегмента.

Компилятор Pascal формирует *сегмент кода*, в котором хранится программа в виде машинных команд, *сегмент данных*, в котором выделяется память под глобальные переменные программы и *сегмент стека*, предназначенный для размещения локальных переменных во время выполнения программы.

Адрес хранится в виде адреса сегмента, уменьшенного в 16 раз (два байта), и адреса смещения (два байта) относительно начала сегмента.

Переменные, которые размещаются в динамической области оперативной памяти (в куче – она же Heap) при помощи указателя, называются *динамическими переменными.*

Указатель может принимать значения, равные всем адресам оперативной памяти, в которые возможна запись данных. Указатель может быть равен стандартному значению **`nil`** (пусто), тогда говорят, что соответствующая переменная в оперативной памяти отсутствует, или же, что указатель не получил значение адреса в оперативной памяти.

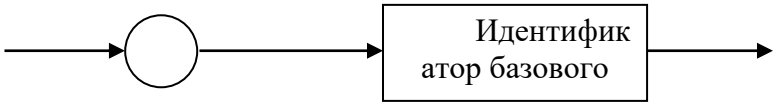
Указатель объявляется при помощи специального символа каре «^», за которым записывается идентификатор типа динамической переменной в соответствии со следующими форматами:

```
type имя_ссылочного типа = тип;
var имя_переменной: имя_ссылочного типа;
```

или

```
var имя_переменной:  ^тип;
```

Ссылочный тип:



К статическим переменным обращение осуществляется по имени, к динамическим – по адресу.

Указатели нужно проинициализировать – присвоить значение.

Получить адрес статического данного можно либо с помощью функции

`Addr(x) : Pointer`,

где `x` – любая переменная, имя процедуры или функции, либо используя унарную операцию @.

Примеры.

```
type
    NameStr  = String [30];
    ArrayInt = array [1 .. 20] of Integer;
var
        RealP    : ^Real;
```

```
                    {указатель на значение типа Real,
                   требующего 6 байт памяти}
        NameStrP : ^NameStr;
        ArrIntP  : ^ArrayInt;
```

Turbo Pascal допускает и такое объявление:

```
type
        PtrType  = ^BaseType;
        BaseType = record
                     x, y : Real;
                     next : PtrType;
                   end;
```

Для других определений здесь была бы ошибка, так как при описании типа `PtrType` идет использование типа `BaseType`, который еще не описан, но для типа «указатель» это возможно.

К статическим переменным обращение осуществляется по имени, к динамическим – по адресу.

Указатели нужно проинициализировать – присвоить значение.

Пример.

```
var
       i     : Integer;
       A     : array [1..10] of Integer;
       P1    : ^Integer;
      Begin
            …
      i := 7;
      P1 := Addr(i);
      {указатель на целое получил адрес переменной i}
         …
       P1 := @A[i];
             {указатель на i-е целое в массиве A}
             {можно P1 := Addr(A[i])            };
       …
       End.
```

Операции

Над значениями ссылочных типов допускаются две операции сравнения: на равенство и неравенство:

```
if P1 <> nil then …
 {проверка, получил ли указатель адрес в памяти}
if P1 = nil then …
 {проверка, ссылается ли этот указатель на nil}
```

Сравнение указателей между собой (P1 = P2) производить не рекомендуется ввиду неоднозначного хранения значения адреса в виде адреса сегмента и адреса смещения.

При сравнении указателей на «<>» или «=» типы, с которыми связаны указатели, должны быть совместимыми. При присваивании значения одного указателя другому типы, с которыми связаны указатели, должны быть совместимыми по присваиванию.

```
var
     P, P1 : ^Integer;
     Pr    : ^Real;
```

то

```
     P1    := nil;  {можно}
     P     := P1;   {можно}
     Pr    := P;    {так нельзя – не совпадает
                      базовый тип}
```

Доступ к переменной по указателю

Обратимся снова к ситуации

```
P1:=@i;
```

Чтобы обратиться к переменной i, есть две возможности:

- использовать идентификатор i;
- использовать адрес этой переменной, который находится в P1.

В последнем случае – при косвенном доступе к переменной через указатель – используют конструкцию, которая называется *разыменование*.

Разыменование означает: для того чтобы по указателю на переменную получить доступ к самой переменной, надо после переменной-указателя поставить знак «^».

После выполнения оператора P1 := @i следующие операторы полностью эквивалентны:

```
i := i+2  и   P1^ := P1^ + 2.
```

Поскольку P1 – указатель на тип Integer, то P1^ – переменная целого типа. А P1^ нужно понимать так: это переменная, на которую ссылается указатель P1.

В результате использования «указателя на указатель» возможно многократное разыменование.

Пример.

```
type   P  =  ^Integer;
var
       P1  : ^Integer;
       PP1 : ^P;
       i   : Integer;
   …
PP1 := @P1;   …   P1 := @i;   …   i := 2;
Write(PP1^^);                    {напечатается 2}.
```

Заметим, что следующая ситуация потенциально опасна:

```
P1 := nil; P1^ := 2;
```

потому что константа **nil** указывает, что не существует никакой динамической переменной.

Действия над динамическими переменными

Основные действия над динамическими переменными – это их создание и уничтожение.

Процедура

```
New(P)
```

предназначена для создания динамической переменной, т. е. для отведения памяти в куче, которую будет использовать динамическая переменная.

При этом динамической переменной отводится блок памяти, который соответствует размеру типа, на который ссылается указатель P (или чуть больше, так как должно быть кратное 8 байтам число), а указателю P выделяется адрес начала блока памяти (адрес динамической переменной).

Если в ходе вычислительного процесса динамическая переменная становится ненужной, ее следует уничтожить процедурой

```
Dispose(P).
```

Эта процедура освобождает память, занятую динамической переменной, и делает значение ее указателя P неопределенным.

При использовании процедуры New надо иметь в виду, что возможен недостаток (исчерпание) памяти для размещения динамической переменной.

В этой ситуации указатель P не получает значения, а программа продолжает выполнение, и никаких сообщений не выдает.

Значит, надо контролировать текущее состояние динамической памяти перед каждым обращением к New.

Функция

```
MaxAvail : Longint
```

возвращает размер (в байтах) наибольшего непрерывного блока в динамической области оперативной памяти, где может быть размещена динамическая переменная.

Функция

```
MemAvail : Longint
```

возвращает суммарный размер (в байтах) свободной области динамической памяти.

Пример.

```
type
       Person  = record … end;
       PPerson = ^Person;
var P : PPerson;
     …
```

```
if MaxAvail >=Sizeof (Person)
then New(P);
{можно разместить и так: P:=New(PPerson)};
    …
 Dispose (P);
```

В приведенном фрагменте программы оператор `New(P)` выделяет память под указатель `P`, который ссылается на переменную типа `Person`, а адрес начала выделенной памяти хранится в `P`. Отметим, что к `New` можно обращаться как к функции: `P:=New(PPerson)`.

Оператор `Dispose (P)` освобождает память, которая связана с указателем `P`, и уничтожает динамическую переменную, и значение указателя `P` становится неопределенным.

Существуют и другие подпрограммы для работы с динамическими переменными.

Процедура

```
GetMem(P, size)
```

создает новую динамическую переменную размером `size` байт, устанавливая значение указателя `P` на начало выделенной ей динамической области (`size` <= 65521).

Процедура

```
FreeMem(P, size)
```

уничтожает динамическую переменную, освобождая `size` байт. После выполнения процедуры значение `P` становится неопределенным.

Процедура

```
Mark(P)
```

запоминает адрес начала распределения данных в динамической памяти, а процедура

```
Release(P)
```

освобождает данные в динамической памяти, начиная с адреса `P`.

Нетипизированные указатели

Стандартный ссылочный тип `Pointer` позволяет не конкретизировать свой базовый тип и совместим со всеми ссылочными типами.

Например, описания

```
var
    {следующие указатели могут указывать на на:}
        P0 : ^Word;      {переменную типа Word    }
        P1 : ^Integer;   {переменную типа Integer }
        P2 : ^Real;      {переменную типа Real    }
        P3 : Pointer;    {на любую переменную     }
```

допускают присваивания значений вида `P3:= P1` или `P3:= P2`.

Значения типа `Pointer` называются нетипизированными (бестиповыми) указателями.

Что адресуется таким указателем – известно лишь программисту.

Если нужно «посмотреть» на динамическую переменную, на которую ссылается P3, как на определенную типизированную, используют механизм приведения типа:

```
Integer (P3^) : = P1^;
Real (P3^)    : = P2^;
Word (P3^)    : = P0^;
```

Указатели дают большие возможности для довольно гибкой работы с памятью, включая адресную арифметику.

Turbo Pascal допускает описание типизированных констант ссылочного типа. Начальным значением таких констант может быть только **nil**. Например:

```
type     Ar   = array [0..5] of Char;
         Ptr  = ^Ar;
const    PP  : Ptr = nil;
var
     A, B  : Ptr;
    {A, B – указатели на переменную - массив символов}
…
```

Для динамических переменных A^, B^ допустимы все те же операции, что и над обычными переменными-массивами.

Программирование алгоритмов с использованием указателей

При разработке программ, выполняющих одинаковые преобразования над большим количеством однотипных данных, используют массивы.

В программе при объявлении каждого статического массива всегда необходимо указывать его размеры.

Это требование делает программы зависимыми от конкретных размеров массивов, что снижает потребительские свойства программ. В таких программах используются динамические переменные-массивы.

Рассмотрим различные подходы при работе с массивами на примере следующих задач.

Задача 1. Выполнить транспонирование матрицы A_{nn} на месте ее размещения.

Задача 2. Упорядочить строки матрицы так, чтобы элементы k-го столбца (k – задается при выполнении программы) образовывали невозрастающую последовательность.

Решение этих задач зависит от того, на каком этапе задаются размеры матрицы и какие это размеры.

Возможны, например, такие ситуации.

1. Известны размеры n, m матрицы A_{nn} (задача 1) и A_{nm} (задача 2), такие, что матрицы поместятся в сегменте данных (в статической памяти).

Эта ситуация не требует распределения матриц в Heap, и их можно объявить, например, так:

1)
```
const
       n=50; {задача 1}
var
       A : array [1..n, 1..n] of Integer;
```
2)
```
const
       n=50; m=30;{задача 2}
var
       A : array [1..n, 1..m] of Integer;
```

Заметим, что при этом действия компилятора следующие: в памяти матрица распределится цепочкой по строкам:

$$A\colon a_{11}, a_{12}, ..., a_{1n}, a_{21}, a_{22}, ..., a_{2n}, ..., a_{n1}, a_{n2}, ..., a_{nn}.$$

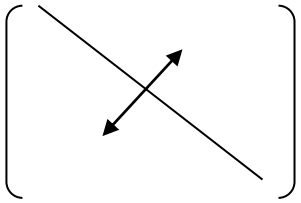
Используем это свойство потом в одном из решений.

2. Если известны размеры *n*, *m* матрицы A_{nn} (задача 1) и A_{nm} (задача 2), такие, что Sizeof($A_{n\times n}$) ≥ 64 кб и Sizeof($A_{n\times m}$) ≥ 64 кб, тогда матрицы необходимо распределить в динамической области частями.

Если размеры *n*, *m* матрицы A_{nn} (задача 1) и A_{nm} (задача 2) задаются во время выполнения программы, тогда матрицы необходимо распределить в динамической области.

Сначала рассмотрим более подробно решение задачи 1.

Схема транспонирования матрицы на своем месте следующая



Отсюда получается такая последовательность действий.

1. Размеры матрицы будем задавать при выполнении программы, учитывая, что Sizeof($A_{n\times m}$) < 64 кб.

2. Матрицу *A* рассматривать как одномерный массив *V*, образованный строками одна за другой.

3. Местоположение элемента a_{ij}, $i=\overline{1,n}$, $j=\overline{1,n}$, будем вычислять относительно начала массива по формуле $k := (i-1)*n+j$.

4. Соответственно местоположение элемента a_{ji}, $i=\overline{1,n}$, $j=\overline{1,n}$, будем вычислять относительно начала массива по формуле $l := (j-1)*n+i$.

5. В виду того, что *V* содержит элементы матрицы *A*, для обмена элементов a_{ij} и a_{ji} нужно выполнить обмен элементов V_k и V_l.

Алгоритм.

1. Объявить матрицу как одномерный массив.
2. Задать значения ее элементов.
3. Напечатать исходную матрицу.

4. Транспонировать матрицу.
5. Напечатать полученную матрицу.

```
program Transp_ukaz;        {Транспонирование}
uses CRT;
type
       TVect  = array [1..1] of Real;
       TPVect = ^TVect;
var
       MasV : TPVect;
       i, n : Byte;
{$R-}          {отключили контроль выхода за пределы
             индексов объявленного массива         }

  procedure Init(var MasV : TPVect; n : Byte);
  var
      i : Integer;
  begin
   for i := 1 to n * n do
           MasV^[i]:= Random*100-50;
  end;

 procedure Show(MasV : TPVect; n : Byte);
  var
     i, j : Integer;
  begin
   Writeln;
   Writeln('MATP');
   for i := 1 to n do
       begin
       for j := 1 to n do
           Write (MasV^[(i - 1) * n + j]:6:0);
       Writeln;
       end;
  end;

 procedure Transp(MasV:TPVect; n:Byte);
  var
       i, j, k, l : Integer;
       r          : Real;
      begin
   for i := 1 to n do
       for j := i + 1 to n do
           begin
 {1}         k       := (i - 1) * n + j;
 {2}         l       := (j - 1) * n + i;
             r       := MasV^[k];
             MasV^[k]:=MasV^[l];
             MasV^[l]:=r;
```

```
            end;
      end;
    begin
      ClrScr;
      Writeln('n – ?');
      Readln(n);
      if MaxAvail < Sizeof(Real)* Sqr(n) then Halt;
      GetMem(MasV, Sizeof(Real)* Sqr(n));
      Init(MasV, n);
      Show(MasV, n);
      Transp(MasV, n);
      Show(MasV, n);
      Readln;
      FreeMem(MasV, Sizeof(Real) * Sqr(n));
    end.
```

Целесообразно было бы определить функцию

```
   function Ord(i, j : Integer) : Integer;
    begin
     Ord := (i - 1) * n + j;
    end;
```

и операторы {1}, {2} заменить вызовом функции `Ord` от соответствующих аргументов: `Ord(i,j)` и `Ord(j,i)`.

Далее рассмотрим решение задачи 2.

Алгоритм.

1. Объявить матрицу.
2. Задать значения ее элементов.
3. Напечатать исходную матрицу.
4. Упорядочить по условию задачи.
5. Напечатать полученную матрицу.

Будем использовать *сортировку выбором*. Находим среди элементов $\{a_{1k}, ..., a_{nk}\}$ наибольший, например a_{pk}. Переставляем *p*-ю строку с первой. Среди оставшихся элементов $\{a_{2k}, ..., a_{nk}\}$, опять находим наибольший. Переставляем строку, в которой он находится, со 2-й строкой.

Продолжаем так и далее, пока не будут сравниваться $a_{n-1,k}$ и a_{nk}. На этом процесс остановится.

Получилась следующая последовательность действий.

1. Образуем цикл по $i = \overline{1, n-1}$.
2. Устанавливаем $p := i$.
3. Образуем цикл по $j = \overline{i+1, n}$.
4. Если $a_{pk} < a_{jk}$, тогда $p := j$.
5. Конец цикла по j.
6. Переставляем *i*-ю и *p*-ю строки.
7. Конец цикла по *i*.

Чтобы написать программную реализацию этого алгоритма, нужно определиться, как будет размещена матрица в памяти. А это зависит не только от ее размера, но и от того, когда (на какой момент) задаются *n* и *m*.

Пусть *n* и *m* – константы и Sizeof(A_{nm}) < 64 кб. Матрицу распределяем в сегменте данных.

```
program Primer_statika;
const
       n=10;     m=10;     k=5;
type
       TItem = Integer;
       TMatr = array[1..n,1..m] of  TItem;
var
       A : TMatr;

procedure Show(A:TMatr; n, m: Byte; Str:String);
{Процедура печати матрицы}
var
     i, j : Byte;
begin
 Writeln(Str:40);
 for I := 1 to n do
     begin
      Writeln;
      for j := 1 to m do
         Write (A[i, j] :5);
     end;
 Writeln;
end;

procedure Init(var A:TMatr; n, m: Byte);
{Процедура инициализации матрицы}
 var
     i, j : Byte;
 begin
  for i := 1 to n do
     for j := 1 to m do
         A[i, j] := Random(101);
 end;
procedure Sort(var A:TMatr; n, m: Byte);
{Процедура сортировки}
  procedure Swap(i, p: Byte);
   {Процедура обмена строк матрицы}
   var
        j : Byte;
        r : TItem;
```

```
   begin
        for j:=1 to m do
            begin
             r       := A[i, j];
             A[i, j] := A[p, j];
             A[p, j] := r
            end;
   end;
var
        i, j, p : Byte;
begin
 for  i := 1 to n-1 do
    begin
     p := i;
     for j := i+1 to  n  do
         if A[p, k] < A[j, k]  then  p := j;
     if p<>i then Swap(i, p);
    end;
 end;
begin
 Randomize;
 Init(A, n, m);
 Show(A, n, m, 'A – исходная');
 Sort(A, n, m);
 Show(A, n, m, 'A – упорядоченная');
end.
```

При решении задачи 2 рассмотрим ситуацию когда *m, n* задаются во время выполнения программы. В этом случае можно использовать разные варианты размещения матрицы в Heap. Рассмотрим их подробнее.

Разные варианты размещения матрицы в Heap

Вариант 1

Рассмотрим ситуацию, когда количество столбцов матрицы *m* – константа, а количество строк становится известным во время выполнения программы (вводится).

Если еще раз проанализировать действия компилятора, то можно заметить, что при выборке элемента a_{ij} матрицы A_{nm} формируется индекс $k := (i-1)*m+j$, по которому в последовательности (по строкам) элементов матрицы и разыскивается нужный элемент.

В следующей программе используем это обстоятельство.

Зададим тип матрицы как бы из одной строки.

При распределении в Heap запросим место на *n* строк процедурой GetMem.

```
program Primer1;
const
         m = 10;     k = 5;
```

```
 type
       TItem  = Integer;
       TMatr  = array [1..1, 1..m] of TItem;      {*}
       TPmatr = ^TMatr;                           {*}
{$R-}
 var
       A : TPMatr;                                {*}
       n : Byte;
procedure Show(A:TPMatr; n, m: Byte; Str:String); {*}
var
     i, j : Byte;
 begin
  Writeln(Str:40);
  for i := 1 to n do
     begin
      Writeln;
      for j := 1 to m do
         Write (A^[i, j] :5);                     {*}
     end;
  Writeln;
 end;
procedure Init(A:TPMatr; n, m: Byte);             {*}
var
     i, j : Byte;
 begin
  for i := 1 to n do
     for j := 1 to m do
         A^[i, j] := Random(101);                 {*}
 end;
procedure Sort(A:TPMatr; n, m: Byte);             {*}
  procedure Swap(i, p: Byte);
   var
        j : Byte;
        r : TItem;
    begin
        for j:=1 to m do
            begin
                r       := A^[i, j];              {*}
                A^[i, j]:= A^[p, j];              {*}
                A^[p, j]:=r                       {*}
            end;
    end;
 var
       i, j, p : Byte;
begin
```

```
 for i := 1 to n-1 do
    begin
     p := i;
     for j := i + 1 to n do
         if A^[p, k] < A^[j, k] then p := j;      {*}
         if p <> i then Swap(i, p);
    end;
 end;
begin
 ClrScr;
 Write ('задай n=');
 Readln (n);
 GetMem (A, n * Sizeof(TMatr));                   {*}
 Randomize;
 Init(A, n, m);
 Show(A, n, m, 'A – исходная');
 Sort(A, n, m);
 Show(A, n, m, 'A – упорядоченная');
 FreeMem (A, n * Sizeof(TMatr));                  {*}
end.
```

Замечание. В данном и последующем вариантах программ существенно следующее: размеры матриц такие, что размер матрицы Sizeof(A_{nm}) < 64 кб.

Вариант 2

Рассмотрим ситуацию, когда количество столбцов m неизвестно и становится известным во время выполнения программы (вводится), а количество строк - константа.

В этом случае нельзя объявить матрицу двумерным массивом.

Разместим матрицу в одномерном массиве (как при решении задачи 1) и сами будем по индексам (`i, j`) отыскивать нужный элемент.

Для этого используем функцию **`function`** `Ord(i,j)`.

Мы существенно будем использовать текст предыдущей программы, все обращения к `A^[i,j]` заменив на `A^[Ord(i, j)]`. Измененные операторы обозначим комментарием `{!}`.

```
program Primer2;
 const
       k : Integer =  5;                          {!}
       n           = 10;                          {!}
                     {k – должна быть типизированной
                      иначе компиляция дает ошибку }
 type
         TItem = Integer;
{$R-}
         TMas  = array [1 .. n*1] of TItem;       {!}
         TPmas = ^TMas;
```

```
 var
      A : TPMas;
      m : Integer;                                    {!}
function Ord (i, j : Byte) : Word;                    {!}
 begin
  Ord := (I - 1) * m + j;
 end;
procedure Show(A:TPMas; n, m: Byte; Str:String);      {!}
 var
     i, j : Byte;
 begin
  Writeln(Str:40);
  for i := 1 to n do
     begin
      Writeln;
      for j := 1 to m do
         Write (A^[Ord(i,j)] :5);                     {!}
     end;
  Writeln;
 end;
procedure Init(A: TPMas; n, m: Byte);                 {!}
 var
     i, j : Byte;
 begin
  for i := 1 to n do
     for j := 1 to m do
         A^[Ord(i,j)] := Random (101);                {!}
 end;
procedure Sort(A:TPMas; n, m: Byte);                  {!}
var
      i, j, p : Byte;
  procedure Swap(i, p: Byte);
   var
        j : Byte;
        r : TItem;
   begin
        for j:=1 to m do
            begin
                r             := A^[Ord(i,j)];        {!}
                A^[Ord(i,j)] := A^[Ord(p,j)];        {!}
                A^[Ord(p,j)] := r                    {!}
            end;
   end;
begin
```

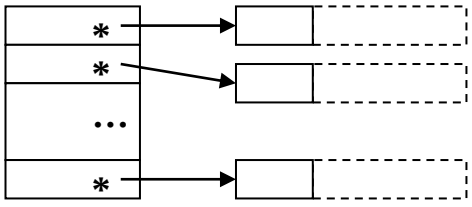
```
 for i := 1 to n - 1 do
     begin
     p := i;
     for j := i + 1 to n do
       if A^[Ord(p,k)] < A^[Ord(j,k)] then p := j; {!}
         if p <> i then Swap(i, p);
     end;
 end;
begin
     Write('input m=');
     Readln(m);
     if MemAvail < m * Sizeof(TMas) then
        begin
        Writeln ('Heap – ?!');
        Halt;
        end;
     GetMem (A, m * Sizeof(TMas));                {!}
     Randomize;
     Init(A, n, m);
     Show(A, n, m, 'A – исходная');
     Sort(A, n, m);
     Show(A, n, m, 'A – упорядоченная');
     FreeMem (A, m * Sizeof(TMas));               {!}
end.
```

Вариант 3.

Рассмотрим ситуацию, когда количество столбцов m неизвестно и становится известным во время выполнения программы (вводится), а количество строк – константа, размеры всей матрицы Sizeof(A_{nm}) > 64 кб, однако строку можно распределить в сегменте.

Количество элементов в строке сразу неизвестно, и будем считать, что в строке есть один элемент.

Заведем массив указателей на каждый из строк матрицы по следующей схеме.



На схеме символом * отмечены данные, значениями которых являются указатели на строки матрицы.

Тогда, когда станет известно количество элементов в строке, нужно будет с каждым указателем на строку `A[i]` связать процедурой `GetMem` в динамической памяти выделенное под строку место.

Обращение к элементу a_{ij} будет реализовываться так:

$$a_{ij} \Rightarrow A[i]^[j].$$

В такой интерпретации матрицы при сортировке строк по фиксированному столбцу, если понадобится поменять местами какие-то две строчки, мы поменяем местами значения ссылок (адреса) на них.

Поэтому выберем другой метод сортировки, а именно: будем искать те элементы в зафиксированном столбце, которые не удовлетворяют условию упорядочения, и менять соответствующие строки местами.

```
program Primer3;
 const
       k : Integer =  5;
       n           = 10;
type
         TItem = Integer;
         TRad  = array[1..1]  of  TItem;              {•}
         PTRad = ^TRad;
{$R-}
         TMas  = array[1..n]  of  PTRad;              {•}
 var
         A    : TMas;                                 {•}
         m, i : Word;                                 {•}
procedure Show(A:TMas; n, m: Word; Str:String);       {•}
var
     i, j : Word;
 begin
   Writeln(Str:40);
   for i := 1 to n do
     begin
      Writeln;
      for j := 1 to m do
         Write (A[i]^[j] :5);                         {•}
     end;
   Writeln;
 end;
procedure Init(var A:TMas; n, m: Word);               {•}
var
     i, j : Word;
 begin
  for i := 1 to n do
     for j := 1 to m do
         A[i]^[j] := Random(101);                     {•}
 end;

procedure Sort(var A:TMas; n, m: Word);               {•}
```

```
var
       i, j, p : Word;
       r       : PTRad;                                    {•}
 begin
  for i := 1 to n - 1 do
    begin
     p:=i;
     for j := i + 1 to n do
         if A[p]^[k] < A[j]^[k] then p:=j;                 {•}
     if p <> i then
       begin
        r    := A[i];                                      {•}
        A[i] := A[p];                                      {•}
        A[p] := r
       end;                                                {•}
    end;
 end;
begin
 Write('input m=');                                         {•}
 Readln(m);                                                {•}
    {Размещение матрицы по строкам }
 for  i := 1  to  n  do                                    {•}
     GetMem(A[i], m * Sizeof(TRad));                       {•}
 Randomize;
 Init(A, n, m);
 Show(A, n, m, 'A – исходная');
 Sort(A, n, m);
 Show(A, n, m, 'A – упорядоченная');
 for  i := 1  to  n  do                                   {•}
     FreeMem(A[i], m * Sizeof(TRad));                     {•}
end.
```

Вариант 4.

Самостоятельно рассмотрите ситуацию, когда количество строк неизвестно и становится известным во время выполнения программы (вводится), а число столбцов – константа, но Sizeof(A_{nm}) > 64 кб.

Здесь можно реализовать задачу так, чтобы работать с транспонированной матрицей, и тогда можно применить предыдущий алгоритм.

Вариант 5.

Рассмотрим ситуацию, когда размеры матрицы неизвестны и становятся известными во время выполнения программы (вводятся).

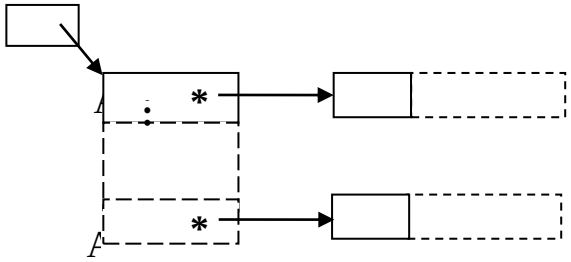
Решение задачи дальше зависит от того, поместится ли матрица целиком в сегмент (Sizeof(A_{nm}) < 64 кб). При положительном ответе решение задачи можно реализовать, представив матрицу как одномерный массив сначала из

одного элемента, а затем, если размеры матрицы станут известными, распределить в динамической памяти всю матрицу. Этот алгоритм мы уже рассматривали в варианте 1.

Предлагаем еще один вариант решения задачи, который будет подходить и в случае, когда Sizeof(A_{nm}) < 64 кб и Sizeof(A_{nm}) > 64 кб. Здесь мы будем использовать подход, изложенный в варианте 3.

Заведем «генеральный» указатель на массив указателей из одного элемента на строку матрицы, которая также состоит из одного элемента.

Тогда схема представления матрицы будет следующей:



и обращение к элементу a_{ij} будет реализовываться так:

$$a_{ij} \Rightarrow \texttt{A\^{}[i]\^{}[j]}.$$

Как и прежде, мы существенно будем использовать текст предыдущей программы. Измененные операторы обозначим комментарием {~}.

```
program  Primer5;
const
         k : Integer =  5;
type
         TItem  = Integer;
         TRad   = array[1..1]  of  TItem;
         TPRad  = ^TRad;                                  {~}
{$R-}
         TMas  = array[1..1]  of TPRad;                   {~}
         TPMas = ^ TMas;                                  {~}
var
         A       : TPMas;                                 {~}
         n, m, i : Word;                                  {~}

procedure Show(A: TPMas; n, m: Word; Str:String);         {~}
var
     i, j : Word;
 begin
  Writeln(Str:40);
  for i := 1 to n do
    begin
     Writeln;
```

```
    for j := 1 to m do
        Write (A^[i]^[j] :5);                          {~}
   end;
  Writeln;
 end;

procedure Init(A: TPMas; n, m: Word);                  {~}
 var
    i, j : Word;
 begin
  for i := 1 to n do
    for j := 1 to m do
        A^[i]^[j] := Random (101);                     {~}
 end;

procedure Sort(A: TPMas; n, m: Word);                  {~}
 var
      i, j, p : Word;
      r       : TPRad;                                 {~}
 begin
  for i := 1 to n - 1 do
    begin
     p := i;
     for j := i + 1 to n do
         if A^[p]^[k] < A^[j]^[k] then p := j;         {~}
         if p<>i then
           begin
            r    := A^[i];                             {~}
            A^[i]:= A^[p];                             {~}
            A^[p]:= r
           end                                         {~}
    end;
 end;
begin
 Writeln('n, m - ??');
 Readln(n, m);
 GetMem(A, n * Sizeof(TMas));                          {~}
 for  i := 1  to  n  do                                {~}
    GetMem(A^[i], m * Sizeof(TPRad));                  {~}
 Randomize;
 Init(A, n, m);
 Show(A, n, m, 'A – исходная');
 Sort(A, n, m);
 Show(A, n, m, 'A – упорядоченная');
 for  i := 1  to  n  do                                {~}
   FreeMem(A^[i], m * Sizeof(TPRad));                  {~}
 FreeMem(A, n * Sizeof(TMas))                          {~}
```

```
end.
```

Проблема потерянных ссылок

При работе с указателями желательно учитывать следующее:

1. Процедуры `Mark(P)` и `Release(P)` фактически должны вызываться в программе один раз, иначе могут быть не спрогнозированные ситуации.

2. Нужно осторожно пользоваться ссылками, объявленными в подпрограммах, т. к. локальные переменные появляются во время вызова подпрограммы, распределяясь в сегменте стека, и исчезают после завершения работы подпрограммы.

3. Нужно стремиться освобождать выделенные области сразу же после того, как необходимость в них отпадает, иначе «загрязнение» памяти ненужными динамическими переменными может привести к быстрому ее исчерпанию.

Проанализируем следующую ситуацию:

```
program LostReference;
type
      TPPerson = ^TPerson;
      TPerson  = record …{информационная часть записи}
                 end;
var
      Memory : Longint;
 procedure GetPerson;
  var  P : TPPerson;
       {локальная переменная – указатель на запись}
  begin
   P := New(TPPerson)
  end;
begin
 Memory := MemAvail;
 Writeln(MemAvail);
 GetPerson;
 Writeln (MemAvail);
 if  Memory = MemAvail  then  Writeln ('OK');
end.
```

Вызов `New` в процедуре `GetPerson` приведет к выделению памяти для динамической переменной типа `Person`, указатель на эту переменную присваивается переменной `P`, которая является локальной. Значит, после выхода из подпрограммы она «потеряется», но память, отведенная в подпрограмме, продолжает существовать, поскольку освободить ее можно только явно, с помощью процедуры `Dispose`, но доступа к переменной уже нет, так как ссылка, которая показывала на переменную, потерялась!

Можно запрограммировать вывод общего объема свободной памяти до и после работы подпрограммы. В данном случае он показал бы, что утеряна какая-то часть памяти.

4. После выполнения New(P) второе New(P) с этим же показателем можно выполнять только тогда, когда после первого New(P) будет освобождена память в Heap, связанная с P.

Последовательность операторов New(P); New(P); приведет к ошибке, так как первая ссылка на динамическую переменную, под которую была отведена память процедурой New(P), потеряется при втором выделении памяти под другую динамическую переменную, но связанную опять же с указателем P.

Нужно придерживаться такого правила: при выходе из блока необходимо или освободить (уничтожить) все созданные в них динамические переменные, или сохранить каким-то образом ссылки на них (например, присвоив эти ссылки глобальным переменным).

5. Бывает и другая ситуация, когда некоторая область памяти освобождена, а в программе остался указатель на эту область.

Рассмотрим следующий пример. Что будет напечатано следующей программой?

```
var
      P : ^Integer;
 procedure X1_Proc;
  var
      i : Integer;
        {локальная переменная размещается в
         стековой памяти, и после завершения процедуры
         эта память снова считается незанятой    }
 begin
  i := 12345;
  P := @i;
  Writeln (P^);
 end;
 procedure X2_Proc;
  var
      j : Integer;
  begin
   j := 7777;
   Writeln (P^);
  end;
begin
 X1_Proc;                {напечатается 12345}
 X2_Proc;                {напечатается 7777 }
   {только такая последовательность вызовов процедур}
end.
```

Замечание. Глобальная ссылочная переменная P в процедуре X1_Proc устанавливается на локальную переменную i, которая была в стеке. После завершения процедуры X1_Proc переменная i исчезает (место в стеке освобождается); указатель P «завис». Если же мы вызвали процедуру X2_Proc,

то локальная переменная `j` (того же типа!) будет распределена на то место в стеке, где находилась ранее переменная `i`. Теперь указатель `P` ссылается на `j`, что и подтверждает результат работы программы.

Введение в связные динамические структуры данных

Рассмотрим динамические структуры данных, количество элементов которых сразу неизвестно, и порядок следования их обусловлен самой совокупностью данных, например список студентов, очередь в магазине и т. д.

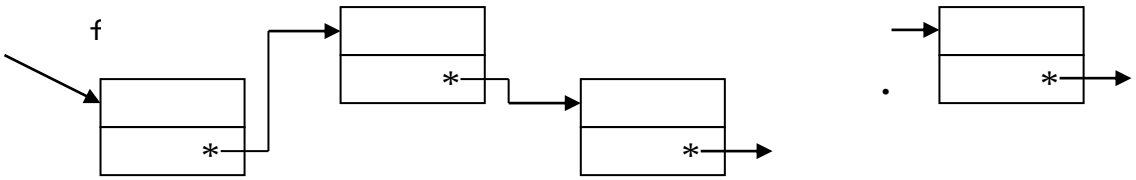
Элементы таких данных последовательно друг за другом можно распределять в `Heap`, связывая их воедино. При таком подходе у элемента должны быть как информационная часть (в дальнейшем на схеме выделена штриховкой), так и ссылочная (на схеме отмечена символом *), которая и поможет включить элемент в последовательность.

Для последовательности линейного вида, количество элементов которых заранее неизвестно, можно применять связную форму хранения их в динамической памяти, например, в виде списка.

Списком называют такую структуру данных, каждый элемент которой содержит ссылку, связывающую его со следующим элементом – звеном (узлом) списка. Звено списка делится на части: информационную и ссылочную.

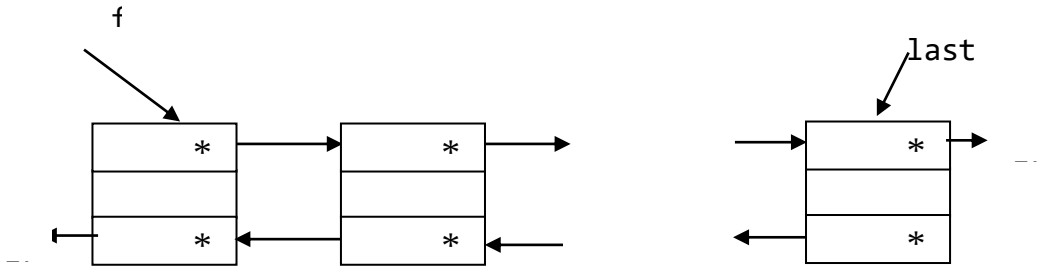
Пусть на первый элемент будет указывать указатель `first`, а на последний – `last`.

Списки бывают разные, в том числе и *однонаправленные линейные*:



По такому списку проход может быть только от `first` (с начала) в конец.

Существуют и *двунаправленные линейные* списки:



На первое звено списка должна быть ссылка, например `first`. А остальные связаны друг с другом собственными связями. Последнее звено в своей ссылочной части ссылается в **`nil`**.

Описание звена списка можно делать в соответствии со следующим фрагментом кода:

```
type
        type_inf =
```

```
            record …
            end;
    Ptr_zveno  = ^zveno;
    zveno      = record
                   inf  : type_inf;
                   next : Ptr_zveno
                 end;
```

Чтобы добавить элемент в список, нужно в Heap под звено запросить память, потом это звено включить в список (первым, в середину, последним), настроив соответствующие связи при помощи ссылочных частей звена.

Алгоритмы работы с линейными списками

Ресурсы для работы со списками лучше объединить в отдельном модуле.
Для работы со списками желательно предусмотреть следующие действия.

1. Размещение элемента списка в динамической памяти.

Можно пользоваться как процедурами New, так и GetMem. например,

```
var    P : Ptr_zveno;
       …
begin
      New(P); …
```

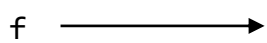
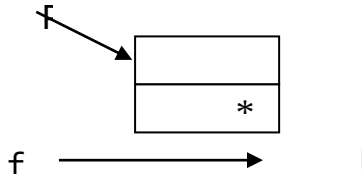
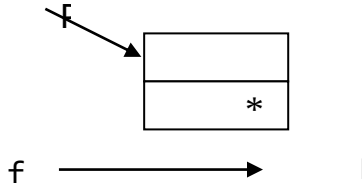
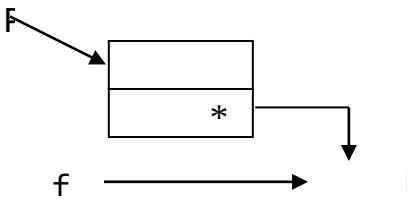
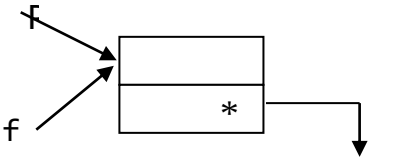
2. Инициализация информационной части звена.

Заполнить информационную часть – значит присвоить значение соответствующим полям записи P^.

3. Включение звена в нужное место списка.

Создание нового списка

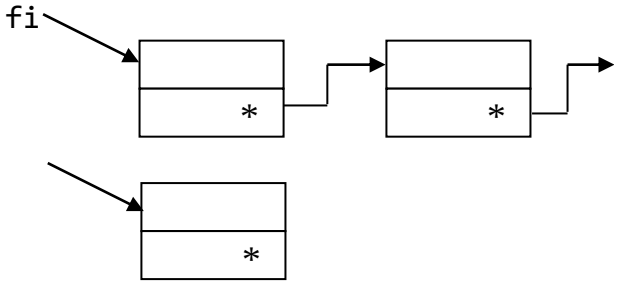
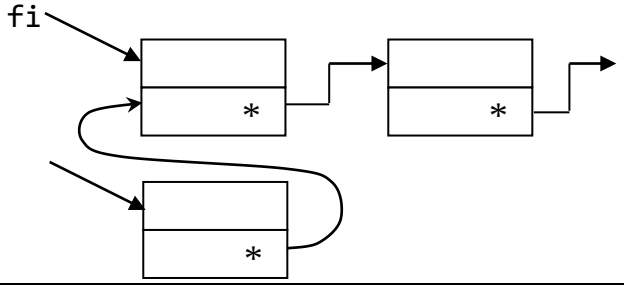
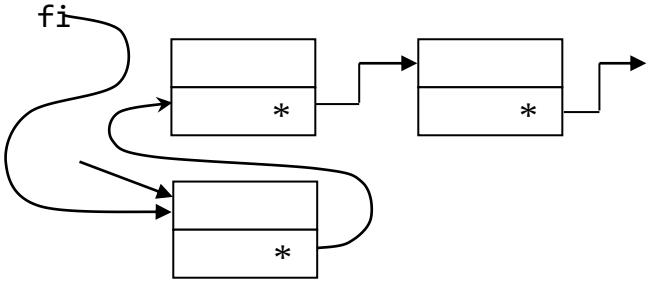
Таблица 25 – Создание нового списка.

Действие	Схема
Первоначально, когда список пустой, указатель `first` устанавливается в **`nil`**. `first:=nil`	
Затем в динамической области создается новое звено – значение типа `Ptr_zveno`. `P := New(Ptr_zveno)`	
Информационная часть звена наполняется информацией при помощи описанной заранее процедуры `Init_inf`. `Init_inf(P)`	
Ссылочная часть нового звена перенаправляется на `first` (фактически на **`nil`**). `P^.next := first`	
Указатель на начало списка `first` перенаправляется на `P`. `first := P`	

Рассмотрим далее, как в список добавить первый элемент, в середину списка и в конец списка.

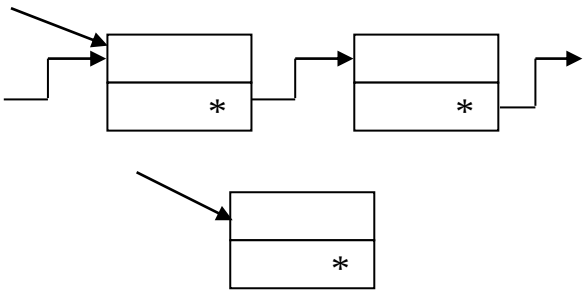
Добавление в начало списка

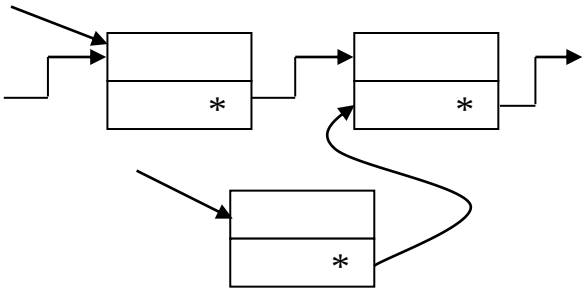
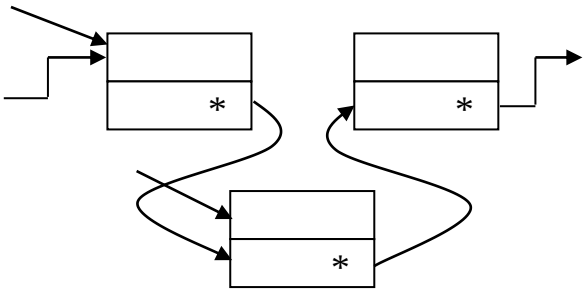
Таблица 26 – Добавление в начало списка.

Действие	Схема
Добавление в начало списка нового звена, на которое ссылается указатель `P`. Исходное состояние	
Ссылочная часть нового звена перенаправляется на `first`. `P^.next := first`	
Указатель на начало списка `first` перенаправляется на `P`. `first := P`	

Добавление в середину списка

Таблица 27 – Добавление в середину списка.

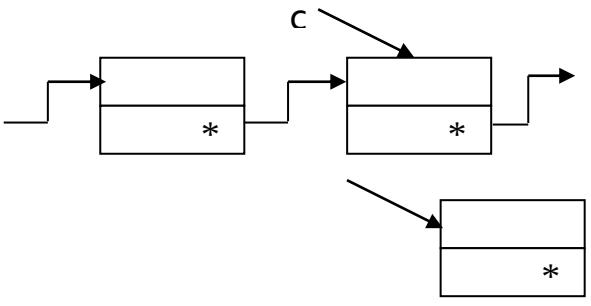
Действие	Схема
Добавление нового звена, на которое ссылается указатель `P`, в середину списка после звена, на которое ссылается указатель `cur`. Исходное состояние	

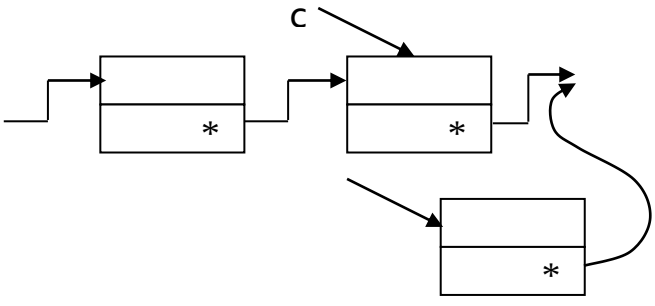
Действие	Схема
Ссылочная часть нового звена перенаправляется на `cur^.next`. `P^.next := cur^.next`	
Ссылочная часть звена, на которое ссылается указатель `cur`, перенаправляется на `P`. `cur^.next := P`	

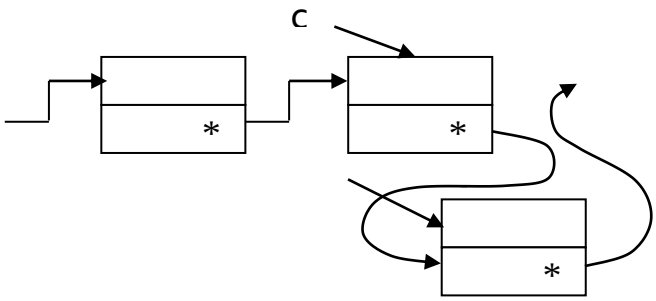
Замечание. Возможно добавление нового звена, на которое ссылается указатель `P`, в середину списка перед звеном, на которое ссылается указатель `cur`. Разработайте такой алгоритм самостоятельно.

Добавление в конец списка

Таблица 28 – Добавление в конец списка.

Действие	Схема
Добавление нового звена, на которое ссылается указатель `P`, в конец списка, после последнего звена, на которое ссылается указатель `cur`. Исходное состояние	

Ссылочная часть нового звена перенаправляется на `cur^.next` (фактически на **`nil`**).
`P^.next :=`
`cur^.next`
	

Действие	Схема
Ссылочная часть звена, на которое ссылается указатель cur, перенаправляется на P. `cur^.next := P`	

4. Поиск элемента списка, на который наложены определенные условия, можно выполнять по аналогии со следующим фрагментом кода.

```
Cur := first;
  while  cur <> nil  do
         begin
              {по информационной части ищем
               нужное звено. Обрабатываем его}
          cur := cur^.next;
          end;
```

5. Распечатка информационной части элемента списка.

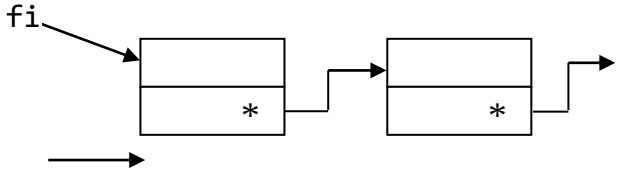
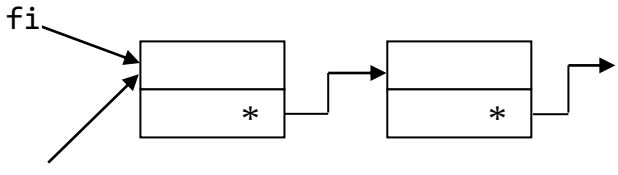
6. Распечатка всех элементов списка.

7. Нахождение и удаление конечного числа элементов списка.

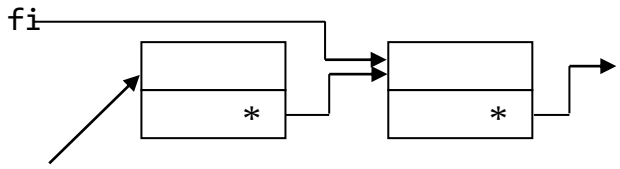
Рассмотрим, как в списке уничтожить первый элемент, элемент в середине списка и последний элемент.

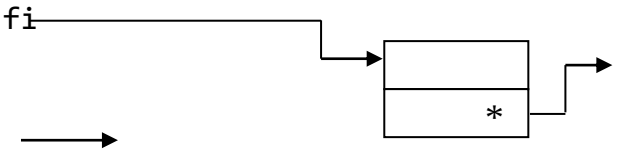
Удаление первого звена в списке

Таблица 29 – Удаление первого звена в списке.

Действие	Схема
Удаление первого звена, на которое ссылается указатель first. Исходное состояние	
Ссылочная переменная P перенаправляется на first. `P := first`	

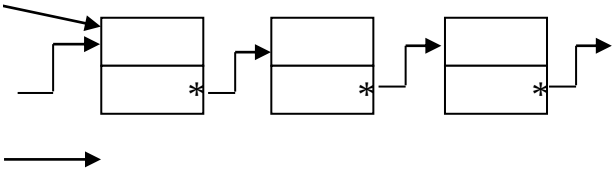
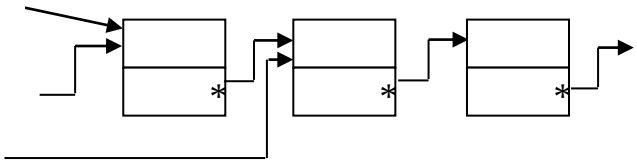
Действие	Схема

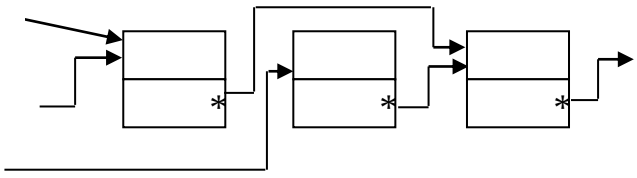
Указатель на начало списка `first` перенаправляется на второй элемент списка.
`first :=`
`first^.next`
	

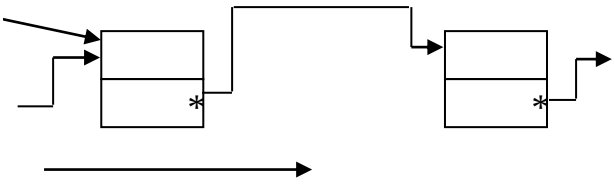
Удаляется звено, которое раньше было первым. `Dispose(P)`	

Удаление звена в середине списка

Таблица 30 – Удаление звена в середине списка.

Действие	Схема
Удаление звена в середине списка, после звена, на которое ссылается указатель `cur`. Исходное состояние	
Ссылочная переменная `P` перенаправляется на `cur^next`. `P := cur^.next`	

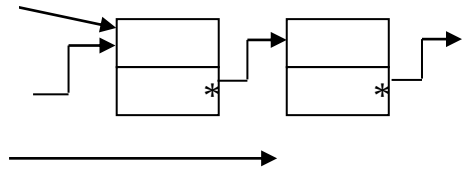
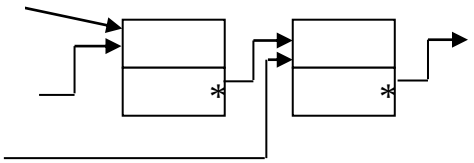
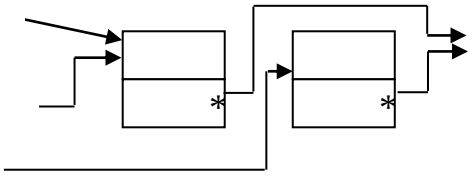
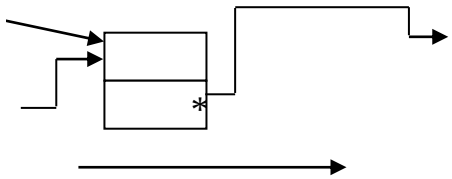
Ссылочная часть звена, на которое ссылается указатель `cur`, перенаправляется на `cur^.next^.next`:
`cur^.next :=`
`cur^.next^.next`
	

Удаляется звено, которое размещено после звена, на которое ссылается указатель `cur`. `Dispose(P)`	

Замечание. Возможно удаление звена, на которое ссылается указатель `cur`, в середине списка. Разработайте такой алгоритм самостоятельно.

Удаление последнего звена из списка

Таблица 31 – Удаление последнего звена из списка.

Действие	Схема
Удаление последнего звена из списка. Указатель `cur` должен ссылаться на предпоследнее звено. Исходное состояние	
Ссылочная переменная `P` перенаправляется на последнее звено `cur^.next`. `P := cur^.next`	
Ссылочная часть звена, на которое ссылается указатель `cur`, перенаправляется на `cur^.next^.next` (фактически на **`nil`**) `cur^.next := cur^.next^.next`	
Удаляется звено, которое расположено после звена, на которое ссылается указатель `cur`. `Dispose(P)`	

8. Сортировка элементов списка.

Схема модуля:

```
unit  Spis_new;
interface
type
      type_inf  = record …  end;
      Ptr_zveno = ^zveno;
      zveno     = record
                        inf  : type_inf;
                        next : Ptr_zveno
                      end;
procedure  Print_inf(P : zveno);
procedure  Init_inf (var P : zveno);
procedure  Sozd_spis(var  first  : Ptr_zveno);
procedure  Show_Spis(const first : Ptr_zveno);
           …
```

```
implementation
…    {В соответствии с предложенными схемами разработайте процедуры самостоятельно}
end.
```

Задание. Для полинома $P_n(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ степени n, в котором отсутствует много одночленов, задаются пары чисел $\{i,\ a_i\}$, причем пара, у которой $a_i = 0$, отсутствует.

Требуется:

- ввести коэффициенты полинома;
- распечатать в виде полинома по спаданию степеней, например в таком виде: $5 * x\text{^}30 - x\text{^}10 + x + 1$;
- подсчитать значение полинома при различных значениях аргумента;
- подсчитать сумму коэффициентов полинома; подсчитать суммы коэффициентов полинома при четных и нечетных степенях.

1.16. Модуль DOS. Модуль CRT

Содержание

- Модуль DOS.
- Модуль CRT. Основные положения.
- Управление звуком.

Модуль DOS

Процедуры и функции модуля DOS предназначены обеспечить удобный способ связи (интерфейс) с программами операционной системы MS DOS. Все они отсутствуют в стандарте языка.

Подключение модуля:

```
uses DOS;
```

Всего в модуле DOS можно насчитать шесть функциональных групп:

- опрос и установка параметров MS DOS;
- работа с датой и временем ПК;
- анализ ресурсов дисков;
- работа с каталогами и файлами;
- работа с прерываниями MS DOS;
- организация субпроцессора и резидентных программ.

Рассмотрим некоторые из них.

Некоторые процедуры и функции модуля DOS

Таблица 32 – Процедуры и функции модуля DOS.

Название	Группа
`GetDate, SetDate, GetTime, SetTime`	Работа с часами и календарем
`DiskSize, DiskFree,`	Анализ ресурсов дисков
`GetFTime, SetFTime, FExpand, FSplit, FindFirst, FindNext, FSearch, GetFAttr`	Работа с каталогами и файлами
`Intr, MsDos`	Работа с прерываниями MS DOS
`Exec`	Организация субпроцессора и резидентных программ

Модуль CRT

Аббревиатура CRT расшифровывается как «электронно–лучевая трубка». В модуле CRT описаны типы, константы, переменные и реализованы специальные процедуры и функции для работы с текстовой информацией на дисплее, позволяющие:

- управлять текстовыми режимами;
- организовывать окна вывода на экране;
- настраивать цвета символов;
- управлять движением курсора.

Кроме того, в модуль включены функции опроса клавиатуры и процедуры управления встроенным в ПК динамиком.

Подключение модуля:

```
uses CRT.
```

Собственно язык Pascal очень прост и лаконичен, стандартные библиотечные подпрограммы же в своей значительной части образуют связь между языковыми средствами Turbo Pascal и функциями операционной системы, а также помогают организовать работу с устройствами компьютеров, совместимых с `IBM PC`.

Управление звуком

В ПК имеется возможность генерировать при помощи встроенного динамика звуковые сигналы частотой от 37 до 32767 Гц. Воспроизводятся только чистые тона без искажений, сила звука не регулируется (при использовании оператора Writeln(^G) воспроизводится звуковой сигнал длительностью 0,25с и частотой 800 Гц).

Для управления частотой звука и его продолжительностью используются стандартные процедуры `Sound`, `NoSound` и `Delay`.

Процедура `Sound (Hz)` включает внутренний динамик, `Hz: Word` задает частоту сигнала, который генерируется динамиком, в герцах. Звучание можно отменить процедурой `NoSound`.

Процедура Delay (Ms) осуществляет задержку в выполнении программы на Ms: Word миллисекунд (хотя, как оказалось время задержки зависит от тактовой частоты процессора).

Пример 1. Программа, имитирующая сигнал звоночка. Логическую функцию keypressed, которая фактически делает паузу до нажатия какой-либо клавиши, рассмотрим ниже.

```
program bell;
uses crt;
var  i:Integer;
 begin
  repeat
   for i:=9 to 12 do
    begin
     sound(i*100);
     delay(20);
    end;
  until keypressed;
  nosound;
 end.
```

Пример 2. Программа, имитирующая звук сирены.

```
program sirena;
uses crt;
var
  i:Integer;
 begin
 repeat
  for i:=300 to 1800 do
   begin
    sound(i);    delay(400);
   end;
  nosound;
  for i:=1800 downto 300 do
   begin
    sound(i);    delay(400);
   end;
   delay(400);  nosound;
   Writeln('Нажми любую клавишу');
 until keypressed;
end.
```

Ввести в программу нехитрые мелодии можно, зная ноты и их частотные эквиваленты в герцах.

Пример 3.

```
program DemoGamma;
```

```
uses crt;
const
   M : array [1..7] of Integer=
             (262, 294, 330, 349, 392, 440, 494);
         {малая октава ноты До, Ре, Ми, Фа, Соль, Ля, Си}
   T : array [1 .. 7] of Integer=
                      (10, 11, 12, 13, 14, 15, 16);
                 {время звучания}
 var
      i, j : Byte;
 begin
 for j := 1 to 100 do
     begin
     for i := 1 to 7 do
         begin
          Sound(M[i]);   Delay(T[i]);
          NoSound
         end;
     end
 end.
```

Изменяя значения элементов массивов M и T, можно добиться хорошей имитации музыкальных произведений.

Для вычисления частоты сигнала имеется формула:

```
Round(440*exp(ln(2)*(akt-(10-nota)/12))),
```

где `akt` – номер одной из восьми октав, причем самая низкая тональная октава имеет номер “–3” ⇒ –3, –2, –1, ..., 3, 4;

`nota` – номер ноты в октаве:

“До” – 1, “До–диез” – 2, “Ре” – 3, ... “Си” – 13.

Работа с клавиатурой

Главным средством ввода информации в ПК является клавиатура. Все клавиши клавиатуры можно разделить на шесть групп:

№	Назначение	Пример
1	Алфавитно–цифровые и знаковые	A .. Z, a ..z, 0 ..9, +, –, *, /, Esc, Tab, Enter, BackSpace, …
2	Функциональные	F1 ..F12
3	Служебные для управления перамещением курсора и редактирования	↑, →, ↓, ←, End, Home, PageUp, PageDown, Del, …
4	Служебно–управляющие	Alt, Ctrl, Shift
5	Служебные – для фиксации регистров	Caps Lock, Scroll Lock, Num Lock, Ins
6	Вспомогательные	Prt_Sc, Pause/Break …

Каждая клавиша имеет свой номер, называемый Scan–кодом (кодом считывания): Esc – 1; 1! – 2 и т. д. Поэтому Shift левый и Shift правый имеют разные Scan–коды.

В системной памяти компьютера находится буфер клавиатуры. Это циклическая очередь из 15 двухбайтовых символов.

Различают три уровня представления и обработки сигналов от клавиатуры: физический, логический, функциональный.

На *физическом* уровне анализируются сигналы, поступающие при нажатии и отпускании клавиш (длительное нажатие воспринимается как многократное).

Клавиатура имеет свой самостоятельный микропроцессор, который по Scan-коду анализирует, какая клавиша нажата (или отпущена), и передает информацию в процессор компьютера.

Клавиши разделяют на три типа:

1. Клавиши и комбинации клавиш, которые после нажатия пересылают в буфер клавиатуры ASCII-код.

Это алфавитно-цифровые клавиши, нажатые одновременно с Shift или без, а также нажатие комбинации клавиш: Ctrl и алфавитно-цифровых; Ctrl и некоторых специальных символов.

2. Клавиши и комбинации клавиш, нажатие которых посылает в буфер клавиатуры расширенный код. Это, например, функциональные клавиши: F1, …, F12; функциональные нажатые одновременно с Shift: Shift+F1, …, Shift+F12; функциональные нажатые одновременно с Ctrl: Ctrl +F1, … Ctrl+F12; функциональные нажатые одновременно с Alt: Alt+F1, …, Alt+F12; алфавитно-цифровые нажатые одновременно с Alt: Alt+ алфавитно-цифровые; управляющие $\downarrow \rightarrow$ … и некоторые другие.

Расширенный код состоит из двух символов: первый символ есть #0, второй – ASCII-код.

3. Клавиши и комбинации клавиш, нажатие которых не посылает в буфер клавиатуры никаких кодов. Состояние их или фиксируется в специальном месте оперативной памяти: это клавиши Shift, Alt, Ctrl, или распознается комбинация клавиш как вызов подпрограммы: например, Ctrl+Alt+Del; Alt+PrintScreen, или игнорируются.

Логический уровень поддерживается базовой системой ввода-вывода. После обработки прерывания, поступившего от клавиатуры, или изменится информация о состоянии нажатия клавиш Shift, Alt, Ctrl, Caps Lock, или в буфер клавиатуры запишется простой или расширенный коды, и управление передается на следующий уровень – функциональный.

На *функциональном* уровне отдельным клавишам программным путем ставятся в соответствие собственные функции, реализуемые при нажатии этих клавиш.

Опрос клавиатуры

Основой алгоритмов управления клавиатуры является анализ в программе буфера клавиатуры. Считывать символы из буфера можно как процедурой Read, Readln, так и функцией модуля CRT

```
Readkey : Char.
```

Команда ch: = Readkey как бы вынимает последовательно введенные в буфер клавиатуры символы по одному за каждое обращение.

Особенности работы функции Readkey:

- полученные функцией символы не отображаются на дисплее;
- режим работы Readkey зависит от состояния буфера ввода:
  - если в буфере есть символы, то функция прочитает очередной символ из буфера (тот, который был введен первым) и удалит его из буфера;
  - если буфер пуст, то функция, а вместе с ней и программа, ожидает ввода символа.

Функция

```
KeyPressed : Boolean
```

возвращает значение true, если в буфере ввода с клавиатуры имеется хотя бы один символ, и false, если буфер пуст.

Следующая группа операторов очищает буфер клавиатуры:

```
while KeyPressed do ch := ReadKey;
```

Можно их оформить процедурой, например, с названием ClrKeyBuf.

Ожидание нажатия клавиши (клавиш), которая вырабатывает простой или расширенный код, осуществляется таким образом:

```
repeat … until KeyPressed.
```

При старте программы буфер обычно пуст.

При работе с клавиатурой возможны следующие ситуации. Если пользователь нажмет комбинацию клавиш:

- может возникнуть и программное прерывание, например (если «система» включила (**on**) анализ этих ситуаций):
  1) Ctrl + Alt + Del;
  2) Ctrl + Break;
  3) Alt + PrtScr;
  4) Ctrl + Alt;
- может образоваться расширенный код;
- комбинация может быть проигнорирована. Если сразу нажать Ctrl, Alt, Shift (последовательно), тогда приоритет имеет Alt, затем Ctrl, затем Shift;
- обычные (символьные) клавиши, даже если они нажаты одновременно с клавишами Ctrl или Shift, выдают символы с ASCII-кодом в диапазоне 1..127 (на русифицированных ПК диапазон до 255). При нажатии клавиш Ctrl + алфавитная эффект определяется ее латинским названием (даже если выбрана русская раскладка клавиатуры): Ctrl + A → 1; … ; Ctrl + Z → 26. Это управляющие символы из таблицы кодов. Клавиши Esc и «пробел» в любой комбинации дают одни и те же коды: 27 и 32 соответственно.

Введенные символы остаются в буфере клавиатуры и доступны для дальнейшей обработки.

Опрос расширенных кодов

Алфавитно-цифровые клавиши, нажатые одновременно с клавишей Alt, и функциональные клавиши посылают в буфер ввода с клавиатуры сразу два символа: первый – #0 (нулевой символ), и второй – с числовым кодом. Вторая часть расширенного кода может совпадать с кодировкой некоторого символа. Так, функциональная клавиша F1 дает расширенный код #0#59; если не учесть, что это расширенный код, тогда код #59 можно трактовать как символ «;».

Подобный механизм значительно повышает информационную отдачу клавиатуры.

Рассмотрим примеры обработки нажатия клавиш клавиатуры.

Пример 1. Программа вывода кодов нажатых клавиш.

```
program PR_key;
uses CRT;
var
    CH : Char;
begin
clrscr;
repeat
      CH := Readkey;
      if CH = #0 then
         begin                              {расширенный код}
         CH := Readkey;
         Write(' спецклавиша ');
         end
         else
             begin
              Write (' Char= ');     {алфавитно-цифровой}
              if CH >= #32 then Write(CH)
              else Write(' ^ ', CHR(ORD(CH) + 64));
             end;              {управляющий символ}
      Writeln (' ASCII= ', ORD(CH));
{т.к. код может совпадать с управляющим, поэтому печатаем номер}
until CH = #27;                              {выход по ESC}
end.
```

Если код клавиши расширенный, тогда надо к буферу клавиатуры обращаться два раза. Опишем функцию, которая будет возвращать символ и знак расширенного кода.

Пример 2.

```
program PR_KeyD;
uses CRT;
const
      ESC    = #27;        DEL  = #83;
      INSKEY = #82;        HOME = #71;
      F1     = #59;        F10  = #68;
```

```
      SF1    = #84;        SF10 = #93;
      CF1    = #94;        CF10 = #103;
      AF1    = #104;       AF10 = #113;
var
     ExtendKey : Boolean;
     CH        : Char;
function GetKey(var ExtendKey : Boolean) : Char;
 var
      ch : Char;
 begin
 ExtendKey := false;
 ch        := Readkey;
 if  ch = #0    then
     begin
     ExtendKey := true;
     ch        := Readkey;
     end;
 GetKey := ch
 end;
begin
ClrScr;
repeat
  ch := GetKey(ExtendKey) ;
  if  not  ExtendKey  then
      Writeln ('символ клавиши с кодом=', Byte(ch))
  else
    case ch of
      SF1..SF10 :
        Writeln('Нажата SHIFT+функциональная клавиша');
      CF1..CF10 :
        Writeln('Нажата CTRL +функциональная клавиша');
      F1 .. F10 :
        Writeln('Нажата функциональная клавиша');
      INSKEY    :
        Writeln('Нажата клавиша «вставка» ');
      DEL :
        Writeln('Нажата клавиша «del» ');
      HOME :
        Writeln('Нажата клавиша «home» ');
      else Writeln('Расширенный код #00+', Byte(ch));
   end;
until ch = ESC
end.
```

За каждой клавишей можно условно закрепить определенную ноту и таким образом и на компьютере имитировать музыкальное устройство.

Пример 3. Имитация музыкального инструмента.

Решение. За цифровыми клавишами 0, 1, … , 8, коды которых #48-#55, закрепим ноты 262, 294, 330, 349, 392, 440, 494, 523. Нажатие любой цифровой клавиши позволит воспроизвести соответствующий звук.

```
program DemoInstrument;
uses CRT;
const
      M : array [1 .. 8] of Integer =
                (262, 294, 330, 349, 392, 440, 494, 523);
var    I  : Integer;
       Ch : Char;
begin
while true do
 begin
  Ch:= Readkey;
  case Ch of
    #48       : Halt;                  {клавиша 0}
    #49.. #55 : I := Ord(ch) – 48;
   else
     Write('клавише звук не назначен', ^G, 'Повторите!');
  end;
  Sound(M[i]);
  Delay(100); {здесь можно repeat until KeyPressed}
  NoSound
 end
end.
```

Управление курсором

Работа процедур Write и Writeln вызывает перемещение курсора по экрану дисплея.

Процедура GOTOXY(x, y) перемещает курсор в позицию x (столбец) и y (строка) относительно текущего окна ($1 \le x \le 80$, $1 \le y \le 25$).

Функции WhereX, WhereY – дают соответственно значение x и y – координат курсора относительно текущего окна.

Функции WhereX, WhereY – дают соответственно значение x и y – координат курсора относительно текущего окна.

```
Write('Курсор находится в столбце', WhereX);
Write('Курсор находится в строке ', WhereY);
```

Пример. Звуковое сопровождение украсило и следующую программу. Что выводит следующая программа?

```
uses crt;
type
    Stroka = String[160];
var
```

```
    vhod   : Stroka;
procedure Gostring(x, y : Byte; inst : Stroka);
var
       st1 : stroka;
       i   : Byte;
procedure Zwon;
 begin  Sound(1000); Delay(50); Nosound;  end;
Begin
 St1:='';
 Clrscr;
 St1:=st1+inst;
 Writeln(st1);
 Writeln;
 for i:=1 to Length(st1) do
  begin
   Delete(st1,1,1);
   Gotoxy(x, y);
   Write(st1);
   Zwon;
   Delay(500);
   DelLine;   {удаляет   с   экрана   строку,   в   которой
           находится курсор}
  Gotoxy(20,10);
  Writeln;
  Writeln(st1);
end;
begin
 Gostring(10, 20,'abcdefghijklmnopqrstuvwxyz');
end.
```

Задание. В зависимости от нажатия клавиш ↓ → ↑ ← заставить двигаться символ *, и при помощи этого движения написать свое имя.

Видеодоступ

Основным устройством для отображения информации, которая вводится и выводится, является *дисплей*. Поддержка его работы осуществляется при помощи соответствующего *адаптера* – специальной платы для подключения его к компьютеру. Изображение на экране формируется в двух основных режимах – текстовом и графическом. В первом – на экран выводятся символы, во втором – изображение выводится как объединение цветовых точек – *пикселов*.

Каждый пиксел, или точка, при цветном режиме может светиться разными цветами, пиксел таких размеров, что промежутки между соседними пикселами почти отсутствуют. Если группа смежных пикселов светится, то они воспринимаются как цельный участок.

Адаптер связывает микропроцессор с дисплеем устройством, которое называется контроллером электронно-лучевой трубки, имеющим в своем

составе: *программируемые порты ввода-вывода, знакогенератор, оперативную электронную память*.

В первых моделях ПК применяли адаптеры MDA – монохромный, CGA –Color Graphics Adapter. Затем MCGA – multyCGA, EGA – Enhanced Graphics Adapter, VGA – Video Graphics Adapter, сейчас существуют SVGA-адаптеры и т. д.

Образ одного экрана хранится в электронной оперативной памяти в закодированном виде. *Страница* – образ экрана в памяти ПК (полная копия экрана), которую можно отобразить мгновенно в любое время.

Существует жесткая взаимосвязь между размером электронной памяти (видеобуфером), разрешением экрана, количеством видеостраниц и «диапазоном» цветов пиксела.

В текстовом режиме под один символ дается поле – матрица определенных размеров. Размеры матрицы зависят от разрешимости экрана (обычно 8×8, 8×14, 8×16, 9×14, 9×16).

Если разрешение экрана 640×200, то выбрано 25 строк и 80 столбцов при матрице 8×8 (640×200 ==25×8×80×8).

Разрешение экрана для VGA-адаптера

Таблица 33 – Разрешение экрана для VGA-адаптера.

Адаптер	Разрешение	Кол-во цветов	Число видеостраниц	Объем видеобуфера
SVGA, VGA	640×200 640×350 640×480	16 16 $>2^8$	4 2 1	256k

В VGA-адаптере видеобуфер – 256 кб и более. Начало видеобуфера зависит от адаптера.

Замечание. В случае многостраничного видеодоступа с каждой страницей связан свой курсор и местоположение его хранится в специальном месте оперативной памяти. Текстовый режим работы дисплея поддерживает модуль CRT, графический – Graph.

Установка текстового режима

Текстовые режимы служат для отображения символов из таблицы кодов ПК (ASCII-кодов) и характеризуются количеством символов в строке и строк на экране.

Процедура

```
TextMode (M),
```

где M – типа Word, переключает текстовые режимы вывода информации на дисплей.

Специально для этой процедуры в модуле CRT определены следующие константы (можно пользоваться как названиями, так и числовыми эквивалентами) (таблица 34):

Таблица 34 – Встроенные константы текстовых режимов.

Значение M		Разрешение	Замечание
0	BW40	40×25	Черно-белый при цветном адаптере
1	CO40 = C40	40×25	Цветной
2	BW80	80×25	Черно-белый при цветном адаптере
3	CO80 = C80	80×25	Цветной
256	Font8x8	80/40×43 80/40×50	Цветной, адаптор EGA Цветной, адаптор VGA

Все остальные числовые значения до 65 535, которые не совпадают ни с одной из указанных констант, включают процедурой `TextMode` режим `C80`.

Константа `Font8×8` используется в адаптерах `EGA` и `VGA`, является дополнительной для первых четырех. Например: `TextMode` (CO80+`Font8×8`) изменит режим разрешимости экрана 80 × 25 на режим 80 × 43 (`EGA`) или 80 × 50 (`VGA`), поскольку константа `Font8×8` обеспечивает построение символов не из матриц 8 × 14 и 8 × 16, а из матриц 8 × 8. Когда основной режим не задан, то по умолчанию берется режим `BW80`.

Процедура `TextMode` очищает экран текущим цветом, устанавливает курсор в левую верхнюю позицию с координатами (1, 1), выполняет некоторые другие действия по наилучшему отображению информации на экран.

Значение установленного режима запоминается в переменной `LastMode` типа `Word`, описанной в `CRT`.

Вывод на экран

Для установки цветов символов, которые выводятся, используется процедура `TextColor(Color)`, где `0≤Color≤15.` Для установки цвета фона – процедура `TextBackGround(Color)`, где `0≤Color≤7.`

Эти процедуры связаны с описанной в модуле `CRT` переменной `TextAttr` типа `Byte`, где фиксируется установка цветов символа и фона, а изменение цветов на экране происходит при выполнении каких-либо операторов работы с экраном, например `Write`, `clrscr` и др.

Если цвет символа совпадает с цветом фона, тогда символ становится невидимым. Так можно «спрятать» на определенное время текст.

Для управления яркостью используются стандартные процедуры `LowVideo` (наименьшая), `NormVideo` (нормальная, восстанавливает первоначальный режим яркости), `HighVideo` (повышенная**)**.

Очистка экрана и управление строками на экране

`ClrScr` полностью очищает экран или окно, и курсор помещается в левый верхний угол (цвет определяется текущим цветом фона).

`ClrEol` удаляет все символы в строке от позиции с курсором до конца строки, при этом можно изменить цвет фона в строке. Курсор остается на месте.

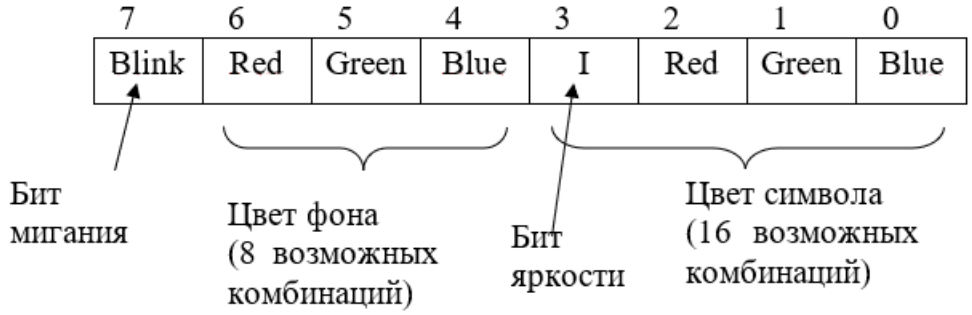
`DelLine` удаляет с экрана строку, в которой находится курсор. Все строки, расположенные ниже, перемещаются на одну строку вверх. В нижнюю часть активного окна добавляется новая строка.

`InsLine` в рамках текущего окна вставляет пустую строку с позиции расположения курсора.

Установка атрибутов цвета символа и фона

Если дисплей работает в текстовом режиме, тогда каждой позиции символа на экране отводится два байта памяти. Первый байт содержит номер кода `ASCII` символа, а второй – атрибуты символа. Цветные адаптеры позволяют выводить в цвете как сам символ, так и фон. Монохромный адаптер ограничен только черным и белым цветами, но он позволяет выводить подчеркнутые символы. Преобразование двух байт, в которых размещены код символа и цвет символа, в пиксельное представление выполняется специальным аппаратным устройством – генератором символов – знакогенератором.

Рассмотрим, как формируется байт атрибута `TextAttr`. По умолчанию устанавливается палитра – набор цветов – на основе цветов: `Red`, `Green`, `Blue`.



`Blink=`2^7`=128`

Основная палитра имеет следующие комбинации цветов (таблица 35):

Таблица 35 – Цвета основной палитры.

Код цвета	Номер цвета в палитре	Название цвета
Основные цвета		
0000	0	Черный (black)
0001	1	Синий (blue)
0010	2	Зеленый (green)
0100	4	Красный (red)
Смесь основных цветов		
0011	3	Циан (бирюзовый) (cyan)
0101	5	Фиолетовый (magenta)
0110	6	Коричневый (brown)
0111	7	Светло-серый (неярко-белый) (LightGray)
Дополнено интенсивностью		
1000	8	Серый (DarkGray)

Код цвета	Номер цвета в палитре	Название цвета
1001	9	Светло-синий (LightBlue)
1010	10	Светло-зеленый (LightGreen)
1011	11	Светло-циан (LightCyan)
1100	12	Светло-красный (LightRed)
1101	13	Светло-фиолетовый (LightMagenta)
1110	14	Светло-коричневый (желтый) (yellow)
1111	15	Ярко-белый (white)

Выводы (таблица 36):

Таблица 36 – Принцип включения цвета палитры.

:

Номер бита	Значение бита	Толкование
0	1	Синий включен в основной цвет
1	1	Зеленый включен в основной цвет
2	1	Красный включен в основной цвет
3	1	Символ выводится с высокой интенсивностью
4	1	Синий включен в основной цвет фона
5	1	Зеленый включен в основной цвет фона
6	1	Красный включен в основной цвет фона
7	1	Символы мигают

Описанные комбинации битов образуют 16 регистров 0-й палитры.

Бывают и другие палитры. Они переключаются при помощи прерываний.

Процедуры установки цвета символа `TextColor` и цвета фона `TextBackGround` связаны с переменной `TextAttr`, но можно сразу менять значение переменной:

`TextAttr:=$1F;` 00011111 – ярко-белый на синем фоне;

`TextAttr:=$71;` 01110001 – синим на белом фоне;

`LowVideo` ⇔ `TextAttr:=TextAttr and $F7` (11110111) (установка в ноль бита яркости и контроль его до отмены).

`HighVideo` ⇔ `TextAttr:=TextAttr or $08` (00001000) (установка в 1 бит яркости).

Вместо процедур:
```
TextColor (Yellow+Blink);
TextBackGround (Red);
```
можно написать: `TextAttr:=Yellow+Blink+Red shl 4;`

Задача 1. Рассмотрим программу вывода 40 окон, координаты которых и цвет фона выбираются случайно.

```
program DemoRandomWindow;
uses Crt;
const  randWx = 80;  randWy = 25;
var    x, y, i : Byte;
```

```
begin
Randomize;
ClrScr;
for i:=1  to  40  do
    begin
    x := succ (Random(randWx));
    y := succ (Random(randWy));
    Window(x, y, x+Random(20), y+Random(8));
    TextBackGround (Random(8));
    ClrScr;
    Write(' Window', ^G, i);
    delay(300);
    end;
Window (1, 1, 80, 25);
ClrScr;
GotoXY (33, 25);
Write ('Автор – я!');
Window (1, 1, 80, 24);
ClrScr;
GotoXY (33, 15);
Write ('Okey!');
delay(300);
ClrScr;                    {останется – "Автор – я!"}
end.
```

Текстовые окна

Модуль CRT поддерживает возможность в любой момент работы программы использовать для вывода определенную часть экрана, так называемое *окно*. Размеры окна задает программист процедурой

```
Window (X1, Y1, X2, Y2),
```

где параметры X1, X2, Y1, Y2 типа Byte. (X1, Y1) – координаты верхнего левого, (X2, Y2) – координаты нижнего правого угла окна.

Если следующие условия не выполняются: $1 \le X1 \le X2 \le X_{max}$ и $1 \le Y1 \le Y2 \le Y_{max}$, то окно создано не будет. После выполнения процедуры Window окно становится текущим. Это значит, что все операции с экраном относятся к той его части, которая определена координатами открытого окна. При этом перемещение курсора происходит только в пределах текущего окна, и позиция с координатами (1, 1) – это левый верхний угол окна.

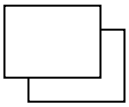
После активизации процедуры Window модуль CRT формирует две специальные переменные WindMin и WindMax типа Word, в которых фиксируются размеры текущего окна (отсчет при этом ведется от (0, 0)). Можно получить значения текущего окна

X1: = Lo (WindMin)+1, Y1: = Hi (WindMin),

X2: = Lo (WindMax)+1, Y2: = Hi (WindMax).

Здесь `Lo` – младший байт младшего слова своего целочисленного аргумента; `Hi` – старший байт младшего слова своего аргумента. Эти координаты фиксируются в специальных местах памяти. Местоположение курсора, его вид также фиксируются в оперативной памяти. Эти переменные `WindMin` и `WindMax` нам нужны, если окно открыто через случайные координаты.

Задание. В середине экрана сделать окно с тенью следующего вида и окантовать его.



Алгоритм. Создаем темное окно (открываем и закрашиваем). Открываем поверху первого второе окно и закрашиваем светлым цветом; стандартная процедура `ClrScr` после открытия окна стирает в видеопамяти всё, что попало «под окно». Окантовываем последнее окно символами псевдографики. Открываем новое третье окно так, чтобы окантовка попала в предыдущее окно.

Задача. Написать программу, которая создает несколько не наложенных окон и затем совершает переход из одного окна в другое.

Решение: Для сохранения координат окон, которые создаются, опишем специальные типы и данные.

```
uses Crt;
type
      WinRecord = record
                  xl, yl, xr, yr : Byte
                  end;
const     maxWin = 3;
type      TMW    = array[1..maxWin]  of  WinRecord;
var       i      : Integer;
const
      MW : TMW = ((xl : 10; yl : 5; xr : 15; yr : 10),
                  (xl : 20; yl : 5; xr : 25; yr : 10),
                  (xl : 30; yl : 5; xr : 35; yr : 10));
begin
ClrScr;
for I := 1  to  maxWin  do
    begin
    with MW[i] do Window(xl, yl, xr, yr);
         ClrScr;
         delay(500);
         end;
    Readln;      {окна созданы по неслучайным координатам.
                           Обращение делаем аналогично.}
    for I := 1  to  maxWin  do
        begin
        with MW[i] do
```

```
                Window (xl, yl, xr, yr);
                delay(5000);
                Writeln ('Old');
                end;
end.
```

Отметим, что на экране находится несколько окон, но в каждый отдельный момент времени активным является только одно, с которым связан курсор. Все процедуры ввода-вывода выполняются относительно активного окна.

1.17. Модуль Graph

Содержание

- Графическое программирование.
- Ресурсы модуля GRAPH.
- Базовые процедуры и функции:
  - управление видеостраницами,
  - перемещение курсора,
  - вывод точки и определение параметров пиксела,
  - вывод отрезка.

Графическое программирование

Стандартное состояние ПК после его включения соответствует работе экрана в текстовом режиме. Любая программа, использующая графические средства компьютера, должна определенным образом инициировать графический режим работы дисплейного адаптера.

Более 80 подпрограмм, содержащихся в модуле `Graph` в файле `GRAPH.TPU`, являются мощными средствами для работы с графической информацией. Подключение модуля стандартное:

```
uses Graph.
```

Настройка графических процедур на работу с конкретным адаптером происходит путем подключения соответствующего графического драйвера. *Драйвер* – это специальная программа, которая осуществляет управление теми или иными техническими средствами ПК. Загрузочные модули драйверов хранятся в специальном файле с расширением `BGI` (`Borland Graphics Interface`). Обычно такие модули записаны в поддиректории `BGI` директории, содержащей `Borland Pascal`.

Для `EGA` и `VGA` адаптеров используется драйвер `EGAVGA.BGI`. В библиотеке модуля `Graph` нет драйверов для новейших адаптеров, и поэтому приходится использовать драйвер `EGAVGA.BGI` и довольствоваться его относительно скромными возможностями.

В графическом режиме экран дисплея условно делится прямоугольной сеткой, каждый элемент которой имеет свои координаты (`x`, `y`). Максимальная величина `x` и `y` зависит от типа дисплея, от драйвера и от режима его работы.

Графическому режиму, как и текстовому, свойственно понятие текущего указателя-курсора, который в графическом режиме невидим.

Образ любого изображения на экране хранится в видеопамяти и восстанавливается примерно через каждые 1/25 с (в текстовом режиме – 1/60с). Там каждому пикселу отводится свой участок (бит, 2 бита, 4 бита, 8 битов и т. д.), где хранится информация о его состоянии и цвете.

Драйвер имеет несколько режимов работы. При инициализации графического режима процедурой `InitGraph` надо указать драйвер, который используется, желательный режим и путь к драйверу.

Если используется драйвер `VGA` (`GraphDriver` = 9), тогда значение режима можно выбирать из следующего множества, приведенного в таблице 37:

Таблица 37 – Значения режимов драйвера `VGA`.

Название режима	Значение константы режима	Разрешение	Количество цветов	Количество видеостраниц
VGA	VGAL0=0	640×200	16	4
	VGAMed=1	640×350	16	2
	VGAHi=2	640×480	16	1

Драйвер можно задавать зарезервированной константой `detect` (= 0), это означает, что при инициализации графического режима будет происходить определение доступного драйвера и установки лучшей разрешимости экрана.

Ресурсы модуля GRAPH

Весь набор стандартных графических подпрограмм можно разделить на такие группы:

- процедуры для подготовки графической системы и переход в текстовый режим;
- опрос текущих режимов;
- процедуры для установки параметров изображения (цвет, стиль заполнения, толщина линии и т. д.);
- процедуры для получения изображения на экране;
- процедуры и функции для получения параметров изображения.

При работе с процедурами и функциями модуля `Graph` могут возникнуть ошибки. Код ошибки возвращается функцией `GraphResult` и соответствует списку констант: `grOk` (= 0) {ошибок нет}, `grNoInitGraph` = –1 графика не инициализирована и т. д. (всего 15 констант).

Процедура `GraphErrorMsg` (`GraphResult`) дает текст ошибки. Запомните, что дважды использовать `GraphResult` для одной и той же операции нельзя – будет плохой результат (флаг ошибок будет сброшен в нуль).

Инициализация и выход из графического режима

Каждый драйвер находится в отдельном файле на диске и содержит выполняемый код и данные. Процедура

```
InitGraph(Graphdriver, Graphmode, Pathdriver)
```

анализирует графическое аппаратное обеспечение, загружает в динамическую память и инициализирует соответствующий графический драйвер, переводит систему в графический режим.

Процедура `CloseGraph` освобождает память от драйвера и восстанавливает предыдущий видеорежим. Процедура `RestoreCrtMode` выполняет переход в текстовый режим, а `SetGraphMode` – в графический. `SetGraphMode` – устанавливает опять по умолчанию все параметры (палитру, текущий указатель, цвет символов, фон и т. д.).

Очистка экрана проще выполняется процедурой `ClearDevice`, а сложнее – `GraphDefaults`. Последняя процедура неявно вызывается при инициализации графического режима.

Пример.

```
program Test;
uses Graph;
var
      GraphDriver : Integer;                    {драйвер}
      GraphMode   : Integer;                    { режим }
      ErrorCode   : Integer;
begin
DirectVideo  :=  false;
GraphDriver  :=  detect;
    {detect   –   для   автоматического   определения   типа
 драйвера аппаратных средств. Если стоит detect, то GraphMode
установится автоматически, иначе надо задавать GraphMode }

InitGraph(GraphDriver, GraphMode, '');
    {если драйвер в текущем каталоге, то путь задается
     пустой строкой, иначе ставим путь поиска библиотеки}

ErrorCode := GraphResult;
       {после опроса этой функции она сбрасывает
        свои результаты в 0. Так как повторный вызов
        функции GraphResult дал бы не тот результат, то мы
          код ошибки сохраняем в переменной ErrorCode }
if ErrorCode <> grOk then            {  Ошибка!}
  begin
    Writeln('Ошибка графики:',GraphErrorMsg(ErrorCode));
```

```
     Writeln ('Программа завершена');
     Halt(1);
    end;
       {*******Работа в графическом режиме*******}
Outtext('  Графический режим. Нажмите <Enter>');
Readln;
       {Задержка экрана, пока не нажмем <ввод> }

RestoreCrtMode;

       {********Работа в текстовом режиме******}
Writeln('Текстовый режим. Нажмите <Enter>');
Readln;
SetGraphMode (GraphMode);
Outtext('Снова графический режим. Нажмите <Enter>');
Readln;
Closegraph;
end.
```

Через соответствующие подпрограммы до или после инициализации графического режима можно получить сведения об установленном режиме:

`GetDriverName` – функция возвращает имя графического драйвера;

`GetGraphMode` – функция возвращает номер графического режима;

`GetMaxMode` – функция возвращает максимальное значение номера режима для текущего загруженного драйвера;

`DetectGraph` – процедура возвращает значение номера текущего драйвера и режима и служит для тестирования графического адаптера;

`GetModeName` – функция возвращает имя заданного графического текущего режима (строка типа '640 × 200 VGA').

Эти процедуры и функции используются для организации диалогового управления графическими режимами.

Базовые процедуры и функции

Рассмотрим далее базовые процедуры и функции в следующей классификации.

- Управление графическими режимами и их анализ.
- Рисование графических примитивов, фигур и перемещение «текущего указателя».
- Управление цветом и заливкой.
- Битовые операции и сохранение графических изображений в памяти.
- Управление страницами.
- Графические окна.
- Управление выводом текста.

Управление видеостраницами

Память адаптеров (видеобуфер) `EGA` и `VGA` имеет более одной видеостраницы. Обычно страницы нумеруются начиная с 0. Для работы с видеостраницами предназначены две процедуры: `SetActivePage` и `SetVisualPage`. Их часто используют при создании анимационных программ.

Процедура

```
SetVisualPage (Page)
```

устанавливает в качестве текущей, т. е. такой, которая отображается на экране, видеостраницу с номером `Page`, а процедура

```
SetActivePage (Page)
```

делает страницу с номером `Page` активной. Это означает, что все графические операции будут происходить на активной странице.

Перемещение курсора

Текущий указатель в графическом режиме невидим, но он присутствует как курсор. Его координаты фиксируется относительно текущей страницы в определенном месте оперативной памяти.

Процедура

```
MoveTo(x, y)
```

перемещает текущий указатель в точку с координатами (x, y). Например, `MoveTo(200, 100).`

Процедура

```
MoveRel(dx, dy)
```

перемещает текущий указатель на `dx` точек по горизонтали и на `dy` – по вертикали. Например, `MoveRel` (5, 10).

Функции `GetX` и `GetY` – возвращают соответственно значения `x-` и `y-` координат текущего указателя. Чтобы координаты не выходили за пределы экрана (в таком случае система не дает ошибки, а отсекает выход), требуется их контролировать функциями

```
GetMaxX : Integer и GetMaxY : Integer,
```

которые возвращают максимально допустимые значения для координат x и y соответственно.

При помощи этих функций можно следующим образом контролировать принадлежность координат точки экрана:

```
if not((x>GetMaxX) or (y>GetMaxY)) then MoveTo(x, y).
```

Для вычисления координат центра получим

```
Xcenter := GetMaxX div 2;
Ycenter := GetMaxY div 2;
```

Вывод точки и определение параметров пиксела

Изображение точки можно получить процедурой

```
PutPixel (x, y, Color),
```

где `x, y : Integer` – заданные координаты, `Color : Word` – цвет (цвет задается по номеру или по названию из таблицы цветов).

Например, изображение точки в центре экрана зеленым цветом получим вызовом процедуры

```
PutPixel (Xcenter, Ycenter, 2).
```

Функция

```
GetPixel(x, y) : Word
```

дает номер цвета пиксела с координатами (x, y).

Вывод отрезка

Процедура

```
Line(x1, y1, x2, y2)
```

рисует отрезок прямой, с началом в точке (x1, y1) и концом в точке (x2, y2) текущим цветом.

Цвет можно предварительно задать процедурой

```
SetColor(Color),
```

где Color задается номером или именованной константой.

Из текущей точки с приращением dx, dy линию чертит процедура

```
LineRel(dX, dY).
```

Из текущей точки в точку с координатами (x, y) линию чертит процедура

```
LineTo(X, Y).
```

Turbo Pascal позволяет чертить линии различного стиля: тонкие, широкие, штриховые, пунктирные и т. д., а также задавать собственный режим вывода. Стиль линии устанавливается процедурой

```
SetLineStyle(LineStyle, Pattern, Thickness);
```

Здесь LineStyle : Word – тип линии, Pattern : Word – образец, Thickness : Word – толщина.

Тип линии LineStyle может принимать значения, которые приведены ниже в таблице 38:

Таблица 38 – Типы линий.

Константа типа линии	Тождественное значение	Описание линии
Solidln	0	Непрерывная
Dottedln	1	Линия из точек
Centerln	2	Штрих-пунктир
Dashedln	3	Штриховая линия
UserBitln	4	Тип программиста

Если определяется тип линии из стандартных типов, тогда Pattern=0; если задается пользовательский, т. е. номером 4, тогда Pattern≠0. В этом случае вторым параметром указывается примитив.

Например, примитив \$5555 (=0101010101010101_2) на месте Pattern показывает, что линия будет перемежаться пробелами и точками по принципу: там, где стоит 0, будет выводиться пиксел цветом фона, а где стоит 1 – цветом линии.

Thickness – толщина линии принимает следующие значения:

NormWidth (= 1) нормальная толщина в 1 пиксел,

ThickWidth (= 3) жирная линия в 3 пиксела.

Если какие-либо параметры, кроме последнего, имеют недопустимые значения, тогда процесс игнорируется, а graphresult принимает значение 11, если же Thickness задан некорректно, тогда толщина берется как NormWidth.

В ряде случаев при использовании Line, LineRel, LineTo и некоторых других процедур требуется устанавливать режим вывода линии на экран (копирование или использование логических операций). Для установки режима вывода кусочно-линейных примитивов предназначена процедура

SetWriteMode(Mode).

Значение параметра Mode определяется стандартными константами:

```
const
      CopyPut = 0;
      XORPut  = 1;
      ORPut   = 2;
      ANDPut  = 3;
      NOTPut  = 4.
```

Если Mode=0, тогда пикселы, расположенные на отрезке прямой линии, переопределяют пикселы на экране, и, таким образом, линия на экране имеет заданный текущий цвет.

Если Mode=1, то пикселы, образующие линию, имеют код цвета, которые образуется операцией **xor** кода текущего цвета и кода цвета пикселов на экране, через которые линия проходит.

Как частный случай такого поведения можно стереть выведенную линию с экрана, исполнив вывод линии еще раз, так как

A **xor** B **xor** B = A.

Для определения текущих характеристик линий служит процедура

GetLineSettings(LST),

где LST : LineSettingsType, а тип LineSettingsType такой:

```
type
     LineSettingsType = record
                          linestyle : Integer;
                          UPattern  : Word;
                          thickness : Integer;
                        end;
```

Управление параметрами образов

Цвета для разных адаптеров

В CGA-адаптерах используется так называемая RGBI-система (Red, Green, Blue, Intensiv) работы с цветом. На базе трех основных цветов путем смешивания и установки различной яркости свечения формируются четыре палитры. Определенный цвет палитры получается путем смешивания трех основных цветов – синего, зеленого и красного. Если для задания цвета служат четыре бита, то три из них используются для указания цветового составляющей: синего (0001), зеленого (0010) и красного (0100). Четвертый бит управляет яркостью свечения: 1000 – высокая интенсивность свечения, 0000 – низкая. По этому принципу формируется 16 цветов, которые поддерживает для адаптера CGA драйвер CGA.BGI. Такую таблицу цветов мы рассматривали в текстовом режиме.

Новые модели видеоадаптеров – EGA / VGA – имеют ряд конструктивных особенностей, позволяющих значительно расширить возможности работы с цветом. Для задания цвета пиксела здесь используются уже шесть битов, а не четыре, как в CGA. Соответственно расширяется гамма цветов. Принцип их формирования примерно то же, но в качестве основы используется система RrGgBb, где RGB – красный, зеленый и синий цвета нормальной яркости, а rgb – те же цвета, но яркость их в 2 раза меньше.

Для EGA/VGA-адаптеров драйвер EGAVGA.BGI устанавливает 64 цвета. Именованные константы для них начинаются словосочетанием EGA.

Основная палитра имеет следующие комбинации цветов (таблица 39):

Таблица 39 – Константы цветов палитры.

Константа	Код цвета	Значение	Название цвета	Компоненты

				цвета
EGABlack	000000	0	Черный	.....
EGABlue	000001	1	Синий	.....B
EGAGreen	000010	2	Зелёный	….G.
EGACyan	000011	3	Голубой	….GB
EGARed	000100	4	Красный	…R..
EGAMagenta	000101	5	Фиолетовый	…R.B
EGABrown	010100	20	Коричневый	.g.R..
EGALightGray	000111	7	Светло-серый	…RGB
EGADarkGray	111000	56	Тёмно-серый	rgb…
EGALightBlue	111001	57	Светло-синий	rgb..B
EGALightGreen	111010	58	Светло-зелёный	rgb.G.
EGALightCyan	111011	59	Светло-голубой	rgb.GB
EGALightRed	111100	60	Светло-красный	rgbR..
EGALighMagentat	111101	61	Светло-фиолетовый	rgbR.B
EGAYellow	111110	62	Жёлтый	rgbRG.
EGAWhite	111111	63	Белый	rgbRGB

Установка цвета

Для различных типов адаптеров количество цветов, которые одновременно отображаются на экране в графическом режиме, может быть разной. Но для всех `BGI`-драйверов она ограничена диапазоном целочисленных значений от 0 до 15. Для того чтобы узнать максимальный номер цвета, который воспринимается данным адаптером в текущем графическом режиме, нужно использовать функцию `GetMaxColor: WORD`. По умолчанию для изображения используется цвет с максимальным номером (т. е. белый), а для фона – с минимальным (черный). Если в процедуре `SetColor(Color)` в качестве `Color` указан недопустимый номер цвета, текущий цвет не меняется. Процедура

```
SetBkColor(Color)
```

устанавливает новый цвет фона, который определяется параметром `Color`.

Получить значения текущих установок цвета можно с помощью двух специальных функций

`GetColor: Word` и `GetBkColor: Word`.

Установка палитры

Палитрой называется максимальный набор цветов, которые поддерживаются драйвером. Она включает 16 цветов (от 0 до 15), которые называются «программными» цветами или цветовыми атрибутами и используются как в текстовом, так и в графическом режимах.

Каждому программному цвету соответствует «аппаратный» цвет из так называемой полной палитры.

В модуле предусмотрен ряд процедур, которые охватывают практически все возможные операции с цветами палитры (таблица 40):

Таблица 40 – Процедуры и функции для работы с палитрой.

Название	Предназначение
`GetDefaultPalette(Palette)`	Получение информации о текущей палитре
`GetPalette(Palette)`	Размещает в переменную Palette информацию о текущей палитре
`SetPaletteSize:Integer`	Возвращает число цветов в текущей палитре
`SetPalette(ColorNum,Color)`	Используется для установки одного цвета палитры
`SetAllPalette(Palette)`	Используется для установки всех цветов палитры

Установка стиля заполнения

Для заполнения внутренних или внешних областей графических фигур в модуле `Graph` встроена группа наперед определенных (стандартных) комбинаций символов-заполнителей, которые можно назвать *маской*. Маска может окрашиваться в допустимые цвета. Комбинацию цвет-маска называют *стилем заполнения* (таблица 41):

Таблица 41 – Стандартные стили заполнения.

Константа	Значение	Маска (заполнение)
`EmptyFill`	`0`	Цвет фона
`SolidFill`	`1`	Текущий цвет
`LineFill`	`2`	Символы – горизонтальные линии
`LtSlashFill`	`3`	Символы наклонные линии// нормальной толщины
`SlashFill`	`4`	Символы // удвоенной толщины
`BkSlashFill`	`5`	Символы \\ удвоенной толщины
`LtBkSlashFill`	`6`	Символы \\ нормальной толщины
`HatchFill`	`7`	Вертикально-горизонтальная штриховка
`XhatchFill`	`8`	Штриховка крест-накрест по диагоналям редкими тонкими линиями
`InterLeaveFill`	`9`	Штриховка крест-накрест по диагоналям частыми тонкими линиями
`WideDotFill`	`10`	Заполнение «частыми» точками
`CloseDotFill`	`11`	Заполнение «редкими» точками
`UserFill`	`12`	Заполнение по определенной пользователем маске заполнения

Процедура

```
SetFillStyle(Pattern, Color)
```

устанавливает стиль Pattern и цвет Color. Значение маски Pattern приведено в предыдущей таблице и может быть задано константой или числом, Color берется из установленной палитры. Например:

```
SetFillStyle(SlashFill, Yellow);
Bar(10, 10, 50, 150);
          {столбец заполнен слешами // желтого цвета}
```

Только когда Pattern = UserFill (Pattern = 12), становится активным шаблон (маска), заданный в SetFillPattern.

Если не подходят предопределенные маски и нужно установить свою, используется процедура

```
SetFillPattern(PattMatrix, Color)
```

PattMatrix : FillPatternType – новая маска, Color – ее цвет. Маска занимает матрицу 8×8 (64 пиксела). Для ее задания используется 8 байт (64 бит), причем каждый бит «зажигает» или «гасит» соответствующий пиксел в матрице 8 × 8. Для этого применяют предопределенный 8-байтовый массив типа

```
type FillPatternType = array[1..8] of Byte;
```

Пример.

```
const
 MyPattern : FillPatternType=
               ($10, $38, $7C, $FE, $7C, $38, $10, $00);
```

А это в двоичной системе счисления имеет вид ромбика:

```
00010000
00111000
01111100
11111110
01111100
00111000
00010000
00000000
```

Необходимую информацию о нестандартных стилях можно получить при помощи процедуры

```
GetFillPattern(inf),
```

которая выгружает в массив Var inf : FillPatternType все содержимое маски.

Процедура

```
GetFillSetting(inf),
```

где inf типа FillSettingType, а

```
type
      FillSettingType = record
```

```
    Pattern : Word;
    Color : Word
  end
```

дает информацию о текущем стиле.

Процедура

```
FloodFill (x, y, border)
```

служит для заполнения внутренней или внешней области фигур (эллипсов, окружностей, многоугольников) при помощи стиля, который был установлен процедурами `SetFillStyle` и `SetFillPattern`.

Если `x`, `y` – координаты внутри фигуры, контур которой имеет цвет `border`, тогда закрашивается внутренняя часть фигуры, а наоборот – выполняется внешнее заполнение текущим образцом зарисовки.

Построение графических фигур

Библиотека `Graph` содержит ряд процедур, формирующих разные фигуры на основе заданных параметров. Цвет, стиль и толщина линии для вычерчивания берутся по умолчанию или устанавливаются соответственно процедурами `SetColor`, `SetFillPattern`, `SetFillStyle`.

Прямоугольники

Изображение контура одномерного прямоугольника дает процедура

```
Rectangle(x1, y1, x2, y2),
```

где `x1, y1` – координаты левого верхнего угла, `x2, y2` – правого нижнего. Область внутри прямоугольника не закрашивается и совпадает по цвету с фоном.

Закрашенный прямоугольник установленным наперед определенным заполнителем и цветом дает процедура

```
Bar(x1, y1, x2, y2).
```

Трехмерный закрашенный прямоугольник (параллелепипед) изображает процедура

```
Bar3D(x1, y1, x2, y2, Delph, Top).
```

Параметр `Delph` – это количество пикселов, которое задает глубину трехмерного контура (чаще всего `Delph := (x2-x1) div 4`). Параметр `Top:Boolean` определяет, строить над прямоугольником вершину (`Top=True`), или нет (`Top=False`).

Многоугольники

Прямоугольники, не ориентированные относительно сторон экрана, можно рисовать разными способами, например, при помощи `Line` или `LineTo`. Но следующая процедура позволяет строить любые многоугольники линией текущего цвета, стиля и толщины:

```
DrawPoly (NumPoints, PolyPoints)
```

Параметр NumPoints: Word задает количество целочисленных координат, которые попарно записываются в нетипизированном параметре переменной PolyPoints.

Замечание. Для вычерчивания замкнутого n-угольника нужно передать (n + 1) координату ((n + 1)-я совпадает с первой).

При выполнении этой процедуры мы будем использовать тип PointType, который введен в модуле Graph следующим образом:

```
type
      PointType = record
                   x, y : Integer;
                  end;
Например,
const
      T : array[1..5] of PointType = ((X :  10; Y :  50),
                  (X : 100; Y :  60), (X : 100; Y : 100),
                  (X :  10; Y : 110), (X :  10; Y :  50));
           …
begin
      DrawPoly (5, T); … end.
```

Будут соединяться первая со второй, вторая с третьей точками и т. д. Чтобы замкнуть фигуру задали массив из пяти элементов и последний элемент равен первому.

Пример. Программа чертит в центре экрана треугольник красными линиями.

```
program DemoDrawPoly;
uses CRT, Graph;
var
      DV, MV              : Integer;
      Pp                  : array [1..4] of PointType;
      xm, ym, ym4, xm4    : Word;
begin
DV      :=  detect;
InitGraph(DV, MV, '');
xm      := GetMaxX;
ym      := GetMaxY;
xm4     := xm div 4;
ym4     := ym div 4;             {координаты вершин}
pp[1].x := xm4;
pp[1].y := ym4;
pp[2].x := xm-xm4;
```

```
pp[2].y := ym4;
pp[3].x := xm div 2;
pp[3].y := ym-ym4;
pp[4]   := pp[1];
SetColor(LightRed);                {цвет для вычерчивания}
DrawPoly(4, pp);
Readln;
CloseGraph
end.
```

При рисовании этой фигуры опросили соответствующими подпрограммами максимальную разрешимость экрана и потом получили образ.

Задание. В центре экрана изобразить окружность случайного радиуса $r \in [10, 20]$ и описать около ее правильный треугольник.

Предыдущий треугольник можно зарисовать, т. е. изменить фон внутри треугольника. Для этого вместо `DrawPoly` используют другую процедуру, но с такими же параметрами:

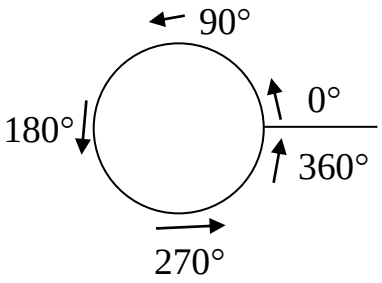
```
FillPoly (NumPoints, PolyPoints)
```

Задача. Нарисовать четырехконечную звезду.

```
program DemoFillPoly;
uses
      CRT, Graph;
const
      Star : array[1..18] of Integer =        {9 пар}
               (75, 0, 100, 50, 150, 75, 100, 100, 75,
                150, 50, 100, 0, 75, 50, 50, 75, 0);
var
      DV, MV     : Integer;
begin
DV           := detect;
InitGraph    (DV, MV, '');
SetFillStyle (1, Green);
FillPoly (9, Star);         {количество пересечений =8+1}
Readln;
CloseGraph;
end.
```

Построение дуг и окружностей

В таких случаях используется полярная система координат.



Процедура

```
Circle(x, y, R)
```

чертит окружность текущим цветом, где (x, y) – центр окружности, R – радиус (все типа Word). Например:

```
SetColor (LightGreen);
Circle (450, 100, 50);
```

Процедура

```
Arc(x, y, St, End, R)
```

чертит дугу окружности, где St, End – начальный и конечный углы (в градусах) дуги окружности с центром в точке (x, y) и радиуса R (если St = 0, End = 360 – получим полную окружность).

Информацию о координатах последнего обращения к Arc дает следующая процедура:

```
GetArcCoords(ArcCoords).
```

Здесь

```
var ArcCoords : ArcCoordsType,

ArcCoordsType = record
                  x, y           : Integer;
                  xStart, yStart : Integer;
                  xEnd, yEnd     : Integer
                end;
```

где (x, y) – координаты центра, (xStart, yStart) – начальная позиция, (xEnd, yEnd) – последняя позиция последней команды Arc.

Эта информация, например, бывает нужной для построения сложных фигур.

Для построения эллиптических дуг предназначена процедура

```
Ellipse (x, y, St, End, xR, yR),
```

где x, y – координаты центра эллипса, xR, yR – длины горизонтальной и вертикальной осей, St, End – начальный и конечный углы (все типа Word). При St=0, End=360 получается полный эллипс. Фон внутри эллипса не изменяется. Например,

```
SetColor (EgaLightCyan);
Ellipse (100, 100, 0, 360, 30, 50);
```

Следующая процедура строит и заливает эллипс:

```
FillEllipse (x, y, xR, yR).
```

Заполнитель устанавливается процедурами `SetFillStyle` или `SetFillPattern`.

В программах деловой графики часто требуется разделить окружность или эллипс на секторы и залить их. Это делается процедурами:

```
PieSlice (x, y, St, End, R)     {сектор в окружности}
Sector (x, y, St, End, xR, yR)     {сектор в эллипсе}
```

Секторы рисуются текущим цветом, а при заливке используются тип и цвет, заданные через `SetFillStyle` и `SetFillPattern`.

Пример. В следующей программе на экране имитируется вид часового циферблата, однако ход часов выбран без связи с системными часами.

```
uses Graph, Crt;
var
    d, r, r1, r2, rr, c, x1, y1, x2, y2, x01, y01, k
                      : Integer;
          Xasp, Yasp : Word;
               XYasp : Real;
begin
d := detect;
InitGraph (d, r, '');
c := GraphResult;
if c <> Grok then Writeln (GraphErrorMsg(c))
   else
   begin
   X1 := GetMaxX div 2;
   Y1 := GetMaxY div 2;
   GetAspectRatio(Xasp, Yasp);     {подсчет радиусов}
   XYasp := Yasp / Xasp;
   r := round(3 * GetMaxY / 8/XYasp);
   r1 := round(0.9 * r);           {часовые деления }
   r2 := round(0.95 * r);          {минутные деления}
   Circle(x1, y1, r);              {циферблат       }
   Circle(x1, y1, round(1.02 * r));
   for k := 0 to 59 do
      begin
       if k mod 5=0 then rr := r1 {часовые деления }
          else rr := r2;          {минутные деления}
       X01 := X1+Round(rr*sin(2*pi*k/60));
```

```
        X2  := X1+Round(r*sin(2*pi*k/60));
        Y01 := Y1-Round(rr*cos(2*pi*k/60)*XYasp);
        Y2  := Y1-Round(r*XYasp*cos(2*pi*k/60));
        Line(X01, Y01, X2, Y2);     {деления}
       end;
   SetWRITEMode(XORPut);
   SetLineStyle(Solidln, 0, ThickWidth);
   r := 0;         {счетчик минут в каждом часе}
   repeat
     for k := 0 to 59 do
     if not KeyPressed then
      begin             {координаты часовой стрелки}
       X2  :=X1+Round(0.85*r1*sin(2*pi*r/60/12));
       Y2  :=Y1-Round(0.85*r1*XYasp*cos(2*pi*r/60/12));
       X01 :=X1+Round(r2*sin(2*pi*k/60));
       Y01 :=Y1-Round(r2*XYasp*cos(2*pi*k/60));
                          {координаты минутной стрелки}
       Line(X1, Y1, X2,  Y2);
       Line(X1, Y1, X01, Y01);
       Delay(1000);
       Line(X1, Y1, X2, Y2);
       Line(X1, Y1, X01, Y01);   {стерли вторым выводом}
       inc(r);
       if r=12*60 then r := 0;
      end;
   until KeyPressed;
   if Readkey=#0 then k := Ord(Readkey);
                              {очистили буфер клавиатуры}
   CloseGraph;
   end
end.
```

Работа с текстом

Задание шрифтов

В комплект поставки Turbo Pascal включается набор штриховых шрифтов. Файлы штриховых шрифтов имеют расширение *.CHR. В штриховых шрифтах при построении символа используется не матричный (как в стандартных шрифтах для текстового режима), а векторный способ. Это дает широкие возможности манипуляции размерами шрифтов без ухудшения качества их изображения. Для доступности штриховых шрифтов нужно, чтобы при инициализации графического режима были доступны файлы с расширением *.CHR, которые их содержат. По умолчанию подключается матричный (битовый) шрифт.

Имена встроенных шрифтов приведены в таблице 42:

Таблица 42 – Имена встроенных шрифтов.

Наперед определенные константы в модуле Graph	Значение	Пояснения
DefaultFont	0	8×8 стандартный битовый шрифт
TriplexFont	1	Штриховые шрифты
SmalFont	2	Малый шрифт
SansSerifFont	3	Шрифт без засечек
GothicFont	4	Готический шрифт

Все эти шрифты содержат только первую часть ASCII-таблицы (0 ÷ 127), и только DefaultFont имеет символы кириллицы (если системно подключена вторая часть таблицы с кириллицей).

Установить нужный шрифт можно процедурой

```
SetTextStyle(Font, Direction, CharSize).
```

Здесь Font : Word – номер выбранного шрифта (из таблицы), CharSize: Word – размер выводимых символов (1≤CharSize≤ 10). Если шрифт матричный (Font = DefaultFont), тогда при CharSize = 4 каждый символ, закодированный матрицей 8 × 8, будет выводиться на основе матрицы 32 × 32 пиксела. Direction : Word – направление:

Horizdir	0	Слева направо
Vertdir	1	Снизу вверх (на 90^{o} повернутый)

Векторные шрифты в основе построения символа реализуют такую идею: в некоторой системе координат описывается последовательность прохождения контура, который образует символ, относительно предыдущей точки контура. Значит, и модификация шрифта (увеличение, расширение и т. д.) происходит простым умножением этих координат на соответствующее число.

Нужный размер шрифта можно установить процедурой

```
SetUserCharSize (multX, divX, multY, divY),
```

где отношения n = multX / divX и m = multY / divY установят ширину и высоту нового шрифта соответственно. Это означает, что, например, при матричном шрифте (8 × 8) каждый пиксел отображается матрицей n×m.

Функции

```
TextHeight (Textstring) : Word  и
TextWidth  (Textstring) : Word
```

дают сведения о высоте и толщине строки Textstring в пикселях. Эта информация нужна, чтобы рассчитать размеры текста для вывода на экран.

Текст выводится относительно текущей позиции (x, y). Текст можно выводить, совмещая координаты вывода с центром строки, или правым, или

левым краем строки, или относительно толщины строки, строку поднять вверх или опустить вниз.

Вариант ориентировки задается процедурой

```
SetTextJustify (Horizontal, Vertical),
```

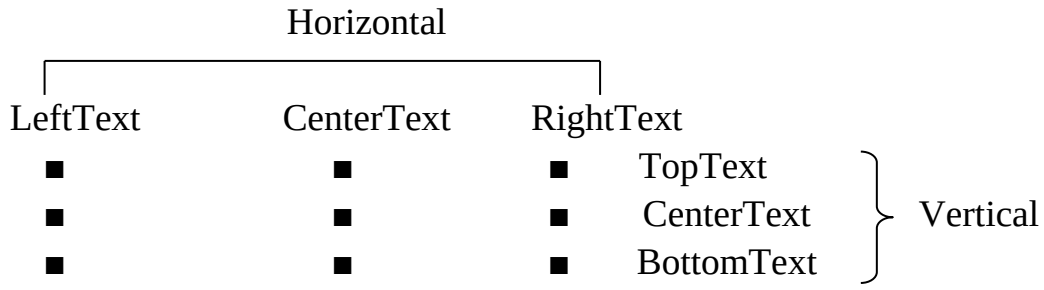
где `Horizontal : Word` содержит значения:

```
0 = LeftText;        1 = CenterText; 2 = RightText;
```

а `Vertical : Word` содержит значения:

```
0 = BottomText; 1 = CenterText; 2 = TopText;
```

На схеме знак ■ показывает координаты точки (x, y)



Вывод текста

Процедура

```
Outtext (TextString)
```

выводит текст `Textstring : String`, начиная с позиции `(x,y)` текущего указателя, а процедура

```
OuttextXY (x, y, Textstring)
```

выводит текст относительно заданных координат `(x,y)` установленным стилем, цветом и ориентировкой. Если текст не помещается в окно, то он отсекается (в случае шрифта `defaultfont` теряется вовсе).

Отметим, что текстовые процедуры `GotoXY`, `Write/Writeln` и установка цвета текста (`TextBackGround`, `TextColor`) в графическом режиме работают, только если переменная `Crt.DirectVideo=false` (или модуль `Crt` не подключен). Ввод `Read/Readln` действует всегда, но при этом текст ввода стирает часть экрана.

Процедура `GettextSettings` дает сведения о шрифте, направлении вывода, размерам символа и ориентации текущего текста.

Для вывода числовых значений предварительно их нужно перевести в строку символов процедурой `Str`.

Пример:

```
Max := 34.56;
Str(Max : 6 : 2, Smax);
OuttextXY (400, 40, 'max='+Smax);
```

Задача. В центре экрана вывести 8 строк разными шрифтами и обвести их рамками.

```
program DemotextFrame;
uses Graph;
var
     Driver, mode  : Integer;
     St, st1       : String;
     Height, width,
     cx, cy, x1, x2,
     y1, y2, i     : Integer;
     tinf          : TextSettingsType;
begin
     St      :=  'AaBbCc…';
     Driver  :=  detect;
     InitGraph(Driver, Mode, '');
     Cx      :=  GetMaxX div 2;
     Cy      :=  GetMaxY div 2;
     SettextJustify(CenterText, CenterText);
     for i := 1 to 8 do
       begin
          Settextstyle (i, 0, 1);
          clearviewport;
          Gettextsettings(tinf);
          str(tinf.font : 2, st1);
          st1    := 'Font : '+st1+'  '+st;
          Height := (Textheight(st1)+4) div 2;
          Width  := (Textwidth (st1)+4) div 2;
          x1     := cx–width;
          x2     := cx+width;
          y1     := cy–height;
          y2     := cy+height;
          SetColor(15);
          Rectangle(x1, y1, x2, y2);
          SetColor(11);
          OuttextXY (CX, CY, st1);
          Readln;
       end;
     Closegraph;
end.
```

Экран и окно

Окно – это прямоугольная область экрана, которая выполняет все функции полного экрана.

После установки окна вся остальная площадь экрана как бы не существует, и весь ввод-вывод осуществляется только через окно. В каждый отдельный момент может быть активным только одно окно. Если окон несколько, за переключение ввода-вывода в нужное окно отвечает программист.

Процедура

```
SetViewPort(x1, y1, x2, y2, Clip)
```

создает окно, где `x1, y1` – координаты левого верхнего угла, `x2, y2` – правого нижнего. Параметр `Clip: Boolean` определяет, будет ли изображение отсекаться при выходе за пределы окна (тогда `Clip := true`), или нет (тогда `Clip := false`). Значит, создав окно, можно получить локальную систему координат и пользоваться отрицательными координатами.

Процедура

```
SetBrColor(Color)
```

задает цвет фона.

Процедура

```
ClearViewPort
```

очищает текущее окно.

Окно существует независимо от текущей страницы и не привязано к странице. По умолчанию сразу окно занимает весь экран, его размеры задаются процедурой инициализации `InitGraph.`

После очистки окна текущий указатель устанавливается в левый верхний пункт окна с координатами (0, 0) – это внутренние координаты окна.

Координатную систему полного экрана можно вернуть, установив новое окно `SetViewPort(0, 0, GetMaxX, GetMaxY, True)`, или выполнить процедуру `GraphDefauls.`

Атрибуты текущего окна можно получить при помощи процедуры

```
GetViewSetting(Vp).
```

Здесь параметр `Vp` описан так: **`var`** `Vp : ViewPortType`, а `ViewPortType` – следующий тип:

```
type
      ViewPortType = record
                     x1, y1, x2, y2     : Integer;
                     Clip               : Boolean;
                     end;
```

Процедура `ClearViewPort` заливает текущим фоном весь экран и поэтому не выделяет границ окна. «Заливки» графического окна цветом, который отличается от общего фона экрана, делают так:

- процедурой SetFillStyle устанавливается шаблон и цвет заполнения;
- процедурой Bar – выводится залитый прямоугольник;
- открывается окно с теми же параметрами, что и в Bar.

Пример.

```
SetFillStyle (1, 4);      {1 – заполнение сплошным цветом,
                           4 – номер цвета Red (красный)}
Bar(100, 50, 500, 200);
SetViewPort(100, 50, 500, 200);
```

Следующая программа под случайное музыкальное сопровождение рисует разноцветные окна.

```
program Demo_SetViewPort;
uses
      Crt, Graph;
var
         I      : Integer;
         DV, DM : Integer;
begin
         DV := detect;
         InitGraph (DV, DM, '');
         SetViewPort (10, 10, 630, 320, True);
         i := 1;
         repeat
                  i := i+1;
                  Sound (Random(180)+40+i);
                  Delay (Random(170));
                  SetFillStyle (Random(4), Random(16));
                  Bar (10, 10, 630, 320);
                  NoSound;
                  Delay(100)
         until KeyPressed;
         Readln;
         CloseGraph;
end.
```

Манипулирование фрагментами образов

Использование оперативной памяти для хранения графических образов, чтобы в дальнейшем быстро восстановить этот образ на экране, полезно, например, для программирования образов, которые движутся.

Процедура

```
GetImage (x1, y1, x2, y2, BitMap)
```

сохраняет в оперативной памяти в переменной BitMap образ прямоугольной области графического экрана:

- (x1, y1) – координаты верхнего левого угла прямоугольной области,
- (x2, y2) – координаты нижнего правого угла.

Размер BitMap должен быть на 4 байта больше, чем необходимо для сохранения образа заданного объема (в этих байтах запоминаются ширина и высота прямоугольной области экрана, которая сохраняется). Переменная BitMap обычно располагается в Heap, где ей предварительно процедурой GetMem отводится память.

Функция

```
ImageSize(x1, y1, x2, y2) : Word
```

возвращает количество байт, необходимых для сохранения прямоугольной области графического образа экрана с координатами (x1, y1), (x2, y2). Если нужно памяти более 64 кб, тогда ImageSize равно 0, а GraphResult = –11, и сохранять образ нужно будет по частям. Величина образа зависит от драйвера и режима, а значит, от количества битов в видеопамяти для одного пиксела.

Процедура

```
PutImage(x, y, BitMap, Mode)
```

выводит на экран сохраненный в BitMap образ прямоугольной области графического образа экрана. Место вывода задается координатами (x, y) – верхний левый угол (а ширина и высота находятся в BitMap). Переменная Mode задает режим вывода на экран. Ее значение соответствует одному из вариантов следующего списка констант:

```
CopyPut   =0;        {MOV}
NormalPut =0;        {MOV}
XORPut    =1;        {xor}
ORPut     =2;        {or}
AndPut    =3;        {and}
NotPut    =4;        {not}
```

Каждая из этих констант соответствует записанной в комментарии бинарной операции между байтами видеопамяти и байтами сохраненного образа. Режим XORPut обеспечивает вывод с видимостью на любом фоне. Повторный вывод с операцией **xor** даст предварительное значение экрана, поскольку A **xor** B **xor** B = A.

Задача. Создать простой вариант движения закрашенного круга, который «падает» с левого верхнего угла в правый нижний.

Решение.

```
program PR_Graph;
uses Crt, Graph;
const
      R  = 20;
```

```
      SX : Word=20;
      SY : Word=20;
var
      GD, GM : Integer;
      P      : Pointer;
      Size   : Word;
begin
       GD := Detect;
       InitGraph (GD, GM, '');
       if GraphResult < > grOk then Halt(1);
       Circle (SX, SY, R);
       FloodFill(SX, SY, GetColor);
                                        {зарисовали круг}
       Size := ImageSize (SX-R, SY-R, SX+R, SY+R);
                          {сколько памяти надо запросить}
       GetMem (P, Size);
             {выделили память для хранения образа в Heap}
       GetImage (SX-R, SY-R, SX+R, SY+R, P^);
                               {записали в буфер в Heap}
       repeat
           Delay(300);
           PutImage (SX-R, SY-R, P^, xorPut);
                     {стираем изображение на старом месте}
           Inc (Sx, 6);
           Inc (Sy, 3);
           PutImage (SX-R, SY-R, P^, XorPut);
                    {пересылаем изображение на новое место}
        until (SX > GetMaxX-2*R) or (SY > GetMaxY-2*R);
        Readln;
        CloseGraph;
        FreeMem (P, Size);
end.
```

Анимация

Первый вариант. Если драйвер обеспечивает работу со страницами графических образов, тогда можно чередовать страницы на экране и получать анимационные фрагменты.

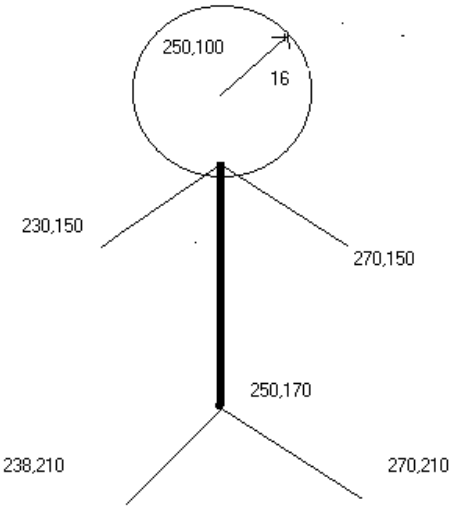
Второй вариант. Выводим рисунок, потом выводим его же цветом, совпадающим с цветом фона, – изображение исчезает, затем таким же методом выводим его в другом месте. Можно при выводе рисунка использовать операцию **xor**. Такой вариант подходит для небольших изображений на одноцветном фоне.

Третий вариант. Выводим изображение, очищаем экран (`ClearViewPort`). Выводим новый образ, очищаем экран и т. д. Такой метод не дает качественных рисунков.

Четвертый вариант. Выводим рисунок, процедурами `PutImage` и `GetImage` какую-то его часть храним в динамической памяти и нужное количество раз восстанавливаем на новых местах.

Простая анимация

Рассмотрим простую анимацию на примере движения конечностей какого-то мультяшного человечка. В системе экранных координат, рассчитаем координаты образа и получим, например, следующую схему для анимации.



Соответственно схеме напишем программу, уточнив некоторые детали.

```
program Demo_Simple_Anim;
uses   Crt, Graph;
const  Ver : array [1 .. 3] of Integer=(150, 112, 76);
       Hor : array [1 .. 3] of Integer=(230, 200, 191);
var       DriverVar,  ModeVar,  i : Integer;
begin
          DriverVar  :=  detect;
          InitGraph (DriverVar, ModeVar, '');
          while not Keypressed do
           begin
              SetColor(15);
              Circle(250, 100, 16);          {     голова}
              Line(250, 117, 250, 170);      {   туловище}
              Line(250, 170, 238, 210);      {левая  нога}
              Line(250, 170, 270, 210);      {правая нога}
              for i := 1 to 3 do
                  begin
                SetColor(15);
```

```
                    Line(250, 120, Hor[i], Ver[i]);
                    Line(250, 120, 250+(250-Hor[i]), Ver[i]);
                    Delay(1000);
                    SetColor(0);
                    Line(250, 120, Hor[i], Ver[i]);
                    Line(250, 120, 250+(250-Hor[i]), Ver[i]);
                     end;
            end;
CloseGraph; end.
```

2. ПРАКТИЧЕСКИЙ РАЗДЕЛ

Программа лабораторных занятий

1 семестр

Тема 1. Арифметика ЭВМ (6 часов).
Переводы чисел. Представление информации в памяти компьютера.
Битовая арифметика.

Тема 2. Алгоритмизация (2 часа).
Схемы алгоритмов: линейные, разветвленные, циклические.
Тестирование алгоритмов, отладка.

Тема 3. Простые данные языка Паскаль и работа с ними (4 часа).
Целочисленные, вещественные. Другие простые. Приведение типов.

Тема 4. Подпрограммы (4 часа).
Процедуры, функции.

Тема 5. Текстовые стандартные файлы (2 часа).
Ввод-вывод.

Тема 6. Базовые операторы в Turbo Pascal (4 + 2 часа).
Операторы простые и структурные.
Операторы if-then-else, case-of-else-end и их использование.
Операторы for-to(downto)-do.
Операторы repeat-until, while-do.

Тема 7. Структуры данных и работа с ними (2 + 2 часа).
Массивы, множества.
Программирование циклов с известным числом повторений.
Программирование циклов с не заданным числом повторений.

Тема 8. Подпрограммы (4 + 2 часа).
Процедуры, функции. Рекурсии и итерации.
Программирование сложных циклов. Эффективность программ.

2 семестр

Тема 9. Записи (2 часа).

Тема 10. Модули пользователя (2 часа).

Тема 11. Файлы (6 + 2 часа).
Создание и обработка типизированных файлов.
Работа с текстовым файлом. Работа с несколькими файлами.
Задача слияния двух отсортированных файлов в третий отсортированный.
Файлы без типа.

Тема 12. Работа с процедурами модуля **SYSTEM** (8 часов).
Использование динамических переменных.

Тема 13. Работа с процедурами модуля **CRT** (6 часов).
Работа с клавиатурой, звуком, окнами в текстовом режиме.

Тема 14. Работа с процедурами модуля **GRAPH** (6 + 2 часа).
Программа BGIDEMO. Создание программ с графикой, анимация.

3. РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ

3.1. Перечень рекомендуемых средств диагностики

Для оценки профессиональных компетенций студентов используется следующий диагностический инструментарий:

– тест – выполнение заданий в тестовой форме;

– коллоквиум – устный опрос на лабораторных занятиях или тестирование;

– текущие контрольные работы.

3.2. Примерный перечень заданий для управляемой самостоятельной работы студентов

В качестве управляемой самостоятельной работы предлагается выполнение тестов.

Тема 1.3. Средства алгоритмического языка Pascal (2 ч)

Базовые элементы языка Pascal. Набор символов, лексемы, разделители.

Форма контроля – компьютерное тестирование на портале *edummf.bsu.by* БГУ в LMS Moodle.

Тема 1.7. Базовые операторы языка и методы программирования (2 ч)

Оператор присваивания. Оператор безусловного перехода, пустой оператор, составной оператор. Условный оператор. Оператор варианта. Операторы повторения. Программирование циклов с известным числом повторений, циклов с предусловием, с постусловием.

Форма контроля – компьютерное тестирование на портале *edummf.bsu.by* БГУ в LMS Moodle.

Тема 1.9. Механизмы структурирования программ (2 ч)

Полное описание процедур и функций. Параметры. Принцип локализации. Рекурсии и итерации. Процедурные типы. Переменные процедурных типов.

Форма контроля – компьютерное тестирование на портале *edummf.bsu.by* БГУ в LMS Moodle.

Тема 2.2. Файлы в языке Pascal (2 ч)

Типизированные файлы. Операции над файлами. Алгоритмы работы с файлами: создание, корректировка, чтение, обработка ошибок ввода-вывода. Текстовые файлы. Нетипизированные (бинарные) файлы.

Форма контроля – компьютерное тестирование на портале *edummf.bsu.by* БГУ в LMS Moodle.

Тема 2.5. Графическое программирование (2 ч)

Базовые процедуры и функции. Управление параметрами изображений. Построение графических примитивов. Работа с текстом. Экран и окно.

Форма контроля – компьютерное тестирование на портале *edummf.bsu.by* БГУ в LMS Moodle.

3.3. Примерный перечень вопросов к зачету

1 курс 1 семестр

1. Структурная методология разработки программ

1. Алгоритм.
2. Основные этапы решения задач на ЭВМ.
3. Тестирование программ.
4. Отладка программ.
5. Структурное программирование и точность программ.
6. Основные конструкции структур управления.
7. Методы разработки программ. *Программирование сверху вниз (от общего к частному). Модульное программирование. Программирование снизу вверх. Структурное кодирование.*

2. Арифметика ЭВМ

1. Системы счисления. *Переводы чисел из одной системы счисления в другую. Перевод целых положительных чисел из системы счисления с основанием "p" в систему счисления с основанием "q". Перевод правильных дробей из системы счисления с основанием "p" в систему счисления с основанием "q". Перевод смешанных дробей.*
2. Формы представления данных. *Формы представления чисел в персональном компьютере. Хранение чисел с фиксированной точкой. Хранение целых чисел. Алгоритм представления отрицательного числа в дополнительном коде. Принципы хранения чисел с плавающей точкой. Форматы чисел с плавающей точкой.*

3. Средства алгоритмического языка PASCAL

1. Общая характеристика алгоритмических языков.
2. Формальное описание алгоритмических языков.
3. Базовые элементы языка Pascal. *Алфавит. Лексическая структура языка.*
4. Общая структура Pascal–программы

4. Простые данные языка Pascal и работа с ними

1. Типы данных.
2. Константы и переменные. *Абсолютные переменные. Типизированные константы.*

3. Целочисленные данные. *Битовая арифметика. Действия битовой арифметики.* Вещественные данные. *Операции над вещественными данными.*
4. Выражения языка.
5. Символьные данные. *Функции.*
6. Логические данные.
7. Данные адресного типа.
8. Данные пользовательского типа.
9. Данные перечислимого типа.
10. Данные интервального типа.

5. Элементарные средства по работе с данными

1. Присваивание значения данным.
2. Простейшее определение процедур и функций. *Параметры.*
3. Файловый тип.
4. Стандартные текстовые файлы.
5. Ввод данных разных типов.
6. Вывод данных разных типов. *Вывод символов. Вывод строковых данных. Вывод логических значений. Вывод целочисленных значений. Вывод данных вещественного типа.*

6. Операторы языка и методы программирования

1. Базовые операторы.
2. Простые операторы. *Пустой оператор. Оператор безусловного перехода GOTO. Оператор вызова процедуры. Составной оператор.*
3. Операторы ветвления. *Условный оператор. Оператор выбора.*
4. Операторы повторения. *Оператор повторения FOR. Оператор повторения REPEAT. Оператор повторения WHILE.*

7. Структуры данных и работа с ними средствами языка Pascal

1. Порядковые типы.
2. Множества. *Типизированные константы типа «множество».*
3. Массивы. *Действия над массивами. Действия над элементами массива. Переменные типа «массив» со стартовым значением. Константы типа массив.*
4. Строки символов. *Присваивание значения строкам. Строковые выражения. Редактирование строк. Преобразование строк.*
5. Комбинированный тип «записи». *Оператор присоединения WITH.*
6. Изменение (приведение) типов и значений

8. Механизмы структурирования программ

1. Процедуры и функции. *Процедуры и функции пользователя.*

2. Параметры. *Параметры ”открытые массивы“. Параметры-значения. Параметры-переменные. Принцип локализации. Побочный эффект.*
3. Рекурсия и итерации.
4. Параметры без типа.
5. Процедуры и функции как параметры. Процедурные типы.
6. Переменные – процедуры и функции

3.4. Примерный перечень вопросов к экзамену

1 курс 2 семестр

1. Файлы в языке Pascal

1. Комбинированный тип «записи». *Записи с вариантами. Оператор присоединения WITH.*
2. Файловые типы.
3. Операции над файлами. *Установочные и завершающие операции. Специальные операции. Операции ввода-вывода для файлов с типом. Последовательный и прямой доступ к файлу с типом.*
4. Практика работы с типизированными файлами. *Обработка ошибок ввода-вывода. Слияние двух отсортированных последовательностей. Создание телефонного справочника.*
5. Текстовые файлы. *Процедуры для работы с текстовыми файлами. Функции для работы с текстовыми файлами.*
6. Файлы без типа

2. Специальные средства языка Turbo Pascal

1. Указатели и динамические структуры данных. *Создание динамических переменных. Операции. Доступ к переменной по указателю. Действия над динамическими переменными. Нетипизированные указатели. Программирование алгоритмов с использованием указателей. Разные варианты размещения матрицы в Heap. Проблема потерянных ссылок. Введение в связные динамические структуры данных. Алгоритмы работы с линейными списками.*
2. Модуль DOS.

3. Стандартные приемы работы с устройствами персонального компьютера

1. Основные положения.
2. Модуль CRT.
3. Управление звуком.
4. Работа с клавиатурой. *Опрос клавиатуры. Опрос расширенных кодов.*
5. Управление курсором.

6. Текстовые окна.
7. Видеодоступ. *Разрешение экрана для VGA-адаптера. Установка текстового режима. Очистка экрана и управление строками на экране. Вывод на экран. Установка атрибутов цвета символа и фона.*

4. Графическое программирование

1. Ресурсы модуля GRAPH. *Инициализация и выход из графического режима.*
2. Базовые процедуры и функции. *Управление видеостраницами. Перемещение курсора. Вывод точки и определение параметров пиксела. Вывод отрезка.*
3. Управление параметрами образов. *Цвета для разных адаптеров. Установка цвета. Установка палитры. Установка стиля заполнения. Коэффициент сжатия.*
4. Построение графических фигур. *Прямоугольники. Многоугольники. Построение дуг и окружностей.*
5. Работа с текстом. *Задание шрифтов. Вывод текста.*
6. Экран и окно.
7. Манипулирование фрагментами образов.
8. Анимация. *Простая анимация.*

5. Модули

4. ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ

4.1. Список рекомендуемой литературы

Основная:

1. Расолько, Г. А. Теория и практика программирования на языке Pascal / Г. А. Расолько, Ю.А. Кремень. - Минск. : Вышэйшая школа, 2022. – 533 с.

Дополнительная:

2. Расолько, Г.А. Теория и практика программирования на Pascal / Г. А. Расолько, Ю.А. Кремень. - Минск.: Вышэйшая школа, 2015.

3. Расолька, Г.А. Pascal: тэорыя і практыка праграмiравання: вучэб.-метад. дапам. / Г. А. Расолько, Ю. А. Кремень. – Мн.: БДУ, 2008.

4. Аляев, Ю. А. Практикум по алгоритмизации и программированию на языке Pascal: учеб. пособие / Ю. А. Аляев, В. П. Гладков, О. А. Козлов. М.: Финансы и статистика, 2004.

4.2. Электронные ресурсы

1. Расолько Г. А., Кремень Ю. А., Кремень Е. В. Методы программирования. Учебная программа учреждения высшего образования по учебной дисциплине для специальности первой ступени высшего образования 1-31 03 01 Математика (по направлениям), направление 1-31 03 01-02 Математика (научно-педагогическая деятельность). № УД-9969/уч 2021г. [Электронный ресурс]. – Режим доступа: https://elib.bsu.by/handle/123456789/270994 – Дата доступа: 24.02.2023.

2. Расолько Г.А. Сборник задач по курсу «Методы программирования и информатика» : практикум. В 2 ч. Ч. I. / Расолько Г.А., Е. В. Кремень, Ю. А. Кремень. – Минск : БГУ, 2020. – 97 с. [Электронный ресурс]. – Режим доступа: https://elib.bsu.by/handle/123456789/248829 – Дата доступа: 24.02.2023.

3. Расолько Г. А., Кремень Е. В., Кремень Ю. А. Методы программирования [Электронный ресурс] : учеб.-метод. пособие. В 2 ч. Ч. 1. Основы теории и практики программирования на Pascal / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень. . – Минск : БГУ, 2022. – 1 электрон. опт. диск (CD-ROM). – ISBN 978-985-881-163-1. [Электронный ресурс]. – Режим доступа: : https://elib.bsu.by/handle/123456789/277935 – Дата доступа: 24.02.2023.

4. Расолько Г. А., Кремень Е. В., Кремень Ю. А. Методы программирования [Электронный ресурс] : учеб.-метод. пособие. В 2 ч. Ч. 2. Теория и практика программирования на Pascal / Г. А. Расолько, Е. В. Кремень, Ю. А. Кремень . – Минск : БГУ, 2022. –Доступ из Интернет [Электронный ресурс]. – Режим доступа: https://elib.bsu.by/handle/123456789/277937 – Дата доступа: 24.02.2023.

5. Расолько Г.А. Задания вычислительной практики по курсу «Методы программирования и информатика» : практикум. В 2 ч. Ч. I / Расолько Г.А., Е. В. Кремень, Ю. А. Кремень. – Минск : БГУ, 2020. – 66 с. [Электронный ресурс]. – Режим доступа: https://elib.bsu.by/handle/123456789/248827 – Дата доступа: 24.02.2023.

4.3. Учебно-методическая карта учебной дисциплины

Номер раздела, темы	Название раздела, темы	Лекции	Лабораторные занятия	Количество часов УСР	Форма контроля знаний
1	2	3	4	5	6
1	**Основы теории и практики программирования на Pascal**	**36**	**30**	**6**	
1.1	Структурная методология разработки программ	8	6		Тренировочный тест
1.2	Арифметика ЭВМ		6		
1.3	Средства алгоритмического языка Pascal	2		2	Тест “ЭВМ и программирование.”
1.4	Введение в систему типов	2			
1.5	Простые данные языка Pascal	6	4		
1.6	Средства по работе с данными	4	4		
1.7	Базовые операторы языка и методы программирования	4	4	2	Тест “Операторы языка”.
1.8	Структуры данных и работа с ними средствами алгоритмического языка	4	2		Тест “Простейшая обработка структур данных”.
1.9	Механизмы структурирования программ	6	4	2	Тест “Процедуры и функции”.
2	**Теория и практика программирования на Pascal**	**34**	**30**	**4**	
2.1	Модули	2	2		Тест “Модули”.
2.2	Файлы в языке Pascal	8	6	2	
2.3	Специальные средства алгоритмического языка	10	8		Тест “Динамические переменные”.
2.4	Стандартные приёмы работы с устройствами IBM-PC	6	6		
2.5	Графическое программирование	8	6	2	Тест “Модули CRT и Graph”.