МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования
«Гомельский государственный университет
имени Франциска Скорины»

Программирование на языке Pascal

Практическое пособие в двух частях

Часть 2

Гомель 2005

УДК 004.43(075.8)
ББК 32.973–018.1я73
П 784

Авторы: Е.А. Ружицкая, доцент, кандидат физико-математических наук; Г.Л. Карасёва, доцент, кандидат физико-математических наук; В.В. Орлов, доцент, кандидат технических наук; В.С.Смородин, доцент, кандидат физико-математических наук; Т.М. Дёмова, ассистент; Т.Г.Богданова, ассистент

Рецензенты: О.И.Еськова, доцент, кандидат технических наук
М.С.Долинский, доцент, кандидат технических наук

Рекомендовано к изданию научно-методическим советом Учреждения образования «Гомельский государственный университет имени Франциска Скорины».

П 784 Программирование на языке Pascal: практическое пособие для студентов математических специальностей университета: В 2ч. Ч.2./ Е.А. Ружицкая, Г.Л. Карасёва, В.В. Орлов, В.С. Смородин, Т.М. Дёмова, Т.Г.Богданова; М-во образов. РБ, Гомельский государственный университет имени Франциска Скорины. – Гомель: ГГУ им. Ф.Скорины, 2005. – 92с.

Во второй части практического пособия рассмотрены основные методы простых сортировок, множества, записи, файлы, динамические структуры и графические возможности языка Pascal. Приведены примеры решения широко распространенных в практике задач.

Пособие ориентировано на самостоятельное изучение и предназначено для студентов математических специальностей университета.

УДК 004.43(075.8)
ББК 32.973–018.1я73

© Ружицкая Е.А., Карасева Г.Л., Орлов В.В.,
Смородин В.С., Демова Т.М.,
Богданова Т.Г., 2005
© Учреждение образования «Гомельский
государственный университет
имени Франциска Скорины», 2005

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 4

МЕТОДЫ СОРТИРОВОК 5
Линейный выбор 5
Линейный выбор с обменом 6
Линейный выбор с подсчетом 7
Парный обмен 8
Метод стандартного обмена (пузырька) 10
Метод просеивания 11
Метод линейной вставки 11

МНОЖЕСТВА 14

ЗАПИСИ 18
Фиксированные записи 18
Записи с вариантами 22

ФАЙЛЫ 29
Типизированные файлы 34
Текстовые файлы 41
Нетипизированные файлы 46

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ 48

ГРАФИЧЕСКИЕ ВОЗМОЖНОСТИ ЯЗЫКА PASCAL 60

ЛИТЕРАТУРА 92

ВВЕДЕНИЕ

Во второй части практического пособия рассмотрены основные методы простых сортировок, множества, записи, файлы, динамические структуры и графические возможности языка Pascal. Пособие содержит теоретический материал, необходимый для выполнения лабораторных работ и заданий по вычислительной практике и примеры решения типовых задач по лабораторным работам. Оно является дополнением к лекционному материалу по курсам «ЭВМ и программирование», «Методы программирование и информатика» и ориентировано на самостоятельное изучение.

Пособие составлено в соответствии с учебными программами курсов «ЭВМ и программирование» для студентов 1 курса специальности 1–31 03 03–02 «Прикладная математика» (научно-педагогическая деятельность) и «Методы программирования и информатика» для студентов 1 курса специальности 1–31 03 01 «Математика», утвержденными научно-методическим Советом Учреждения образования «Гомельский государственный университет имени Франциска Скорины».

Практическое пособие предназначено для студентов первого курса математического факультета, изучающих язык программирования Pascal.

Учебное издание

Ружицкая Елена Адольфовна
Карасёва Галина Леонидовна
Орлов Владимир Васильевич
Смородин Виктор Сергеевич
Дёмова Тамара Максимовна
Богданова Татьяна Григорьевна

Программирование на языке Pascal.
Практическое пособие в двух частях. Часть 2.

В авторской редакции

Подписано в печать 01.10.2005 г. (№132). Формат издания 60x84 1/16.
Бумага писчая №1. Печать на ризографе. Гарнитура Таймс.
Усл.п.л. 5,4. Уч-изд.л. 4,3. Тираж 20 экз.

Учреждение образования
«Гомельский государственный университет
имени Франциска Скорины»,
246019 г.Гомель, ул. Советская, 104

Отпечатано в учреждении образования
«Гомельский государственный университет
имени Франциска Скорины»,
246019 г.Гомель, ул. Советская, 104

ЛИТЕРАТУРА

1. Бородич Ю.С. и др. Паскаль для персональных компьютеров: Справ. Пособие / Ю.С.Бородич, А.Н.Вальвачев, А.И.Кузьмич. – Мн.: Выш. шк.: БФ ГИТМП «НИКА», 1991. – 365 с.
2. Вальвачев А.Н., Крисевич В.С. Программирование на языке Паскаль для персональных ЭВМ ЕС: Справ. пособие. – Мн.: Выш.шк., 1989. – 223 с.: ил.
3. Офицеров Д.В. и др. Программирование на персональных ЭВМ: Практикум: Учеб. Пособие / Д.В.Офицеров, А.Б. Долгий, В.А.Старых; Под общ. ред. Д.В.Офицерова. – Мн.: Выш.шк., 1993. – 256 с.
4. Немнюгин С.А. Turbo Pascal: практикум – СПб: Питер, 200. – 256 с.:ил.
5. Пантелеева З.Т. Графика вычислительных процессов: Учеб.пособие. – М.: Финансы и статистика, 1983. – 167 с., ил.
6. Фаронов В.В. Турбо Паскаль 7.0. Начальный курс. Учебное пособие. – М.: «Нолидж», 1997. – 616 с., ил.
7. Фигурнов В.Э. IBM PC для пользователя. Изд. 7-е, перераб. И доп. – М.: ИНФРА – М, 1997. – 640 с.: ил.

МЕТОДЫ СОРТИРОВОК

Сортировка – это процесс расстановки элементов «в некотором порядке».

Различают 3 группы простых сортировок:

1. Методы, основанные на выборе:
   - линейный выбор;
   - линейный выбор с обменом;
   - линейный выбор с подсчетом.
2. Сортировка обменом:
   - парный обмен;
   - метод стандартного обмена (пузырька);
   - метод просеивания.
3. Метод линейной вставки.

Линейный выбор

Для сортировки исходного вектора *A*, содержащего *n* элементов, необходимо *n* раз просмотреть элементы исходного вектора и сформировать полученный упорядоченный вектор *B*. При каждом просмотре находится минимальный элемент вектора и помещается в упорядоченный список.

При первом просмотре находим минимальный элемент вектора А и помещает его в первую позицию вектора *B*. В исходном векторе минимальный элемент заменяется фиктивной величиной, заведомо большей всех элементов вектора.

При втором просмотре опять находим минимальный элемент вектора *A* и помещаем его во вторую позицию вектора *B*. В исходном векторе *A* минимальный элемент снова заменяется фиктивной величиной, заведомо большей всех элементов вектора.

После *n* просмотров исходный вектор *A* будет состоять из фиктивных величин, а вектор *B* – из упорядоченных элементов исходного вектора *A*.

Просмотр	Исходный вектор *A*	Полученный вектор *B*
1-ый	*2 4 8 5 6 1* *2 4 8 5 6 99*	*1*
2-ой	*2 4 8 5 6 99* *99 4 8 5 6 99*	*1 2*
3-ий	*99 4 8 5 6 99* *99 99 8 5 6 99*	*1 2 4*

4-ый	*99 99 8 5 6 99* *99 99 8 99 6 99*	*1 2 4 5*
5-ый	*99 99 8 99 6 99* *99 99 8 99 99 99*	*1 2 4 5 6*
6-ой	*99 99 8 99 99 99* *99 99 99 99 99 99*	*1 2 4 5 6 8*

```
{Исходный вектор A}
{Полученный вектор B}
For i:=1 to n do
begin
    min:=A[i];
    i_min:=i;
    For j:=1 to n do
        If A[j]<min then
        begin
            min:=A[j];
            i_min:=j;
        end;
    B[i]:=min;
    A[i_min]:=99;
end;
```

Для сортировки вектора по убыванию нужно находить максимальный элемент и заменять его значением, заведомо меньшим всех элементов вектора.

Линейный выбор с обменом

Этот метод состоит из *n*-1 просмотра элементов исходного вектора.

При *первом* просмотре находим минимальный элемент вектора и меняем его местами с первым элементом.

При *втором* просмотре первый элемент исключаем из рассмотрения и в оставшемся векторе находим минимальный элемент. Затем меняем его местами со вторым элементов вектора, и.т.д.

После *n*-1 просмотра вектор будет упорядочен.

Просмотр	Исходный вектор *A*
1-ый	*2 4 8 5 6 1* *1 4 8 5 6 2*

```
For i:=1 to n-1 do
begin
  min:=A[i]; i_min:=i;
```


(*x,y*) будем находить, используя параметрическое задание уравнения окружности со смещением на заданный угол:

$$\begin{cases} x = \cos(i \cdot t) \\ y = \sin(i \cdot t) \end{cases}.$$

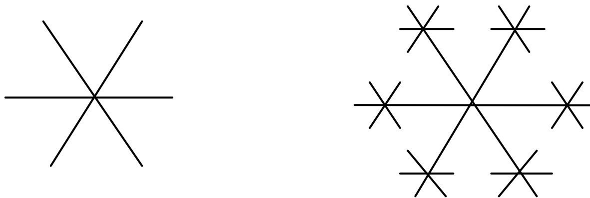
Чтобы величины $\cos(i \cdot t)$ и $\sin(i \cdot t)$ не вычислялись в цикле многократно, запомним их значения в массивах C и S.

```
Program Sneg;
  Uses Graph;
  Const
    k=6;          {количество кристаллов}
    n=5;          {глубина рекурсии}
    t=2*Pi/k;     {угол поворота}
  Var
    Driver,Mode,I :Integer;
    C,S           :array[1..k] of Real;
Procedure Snow(x0,y0,r,m:Integer);
  Var
     x,y,i:Integer;
Begin
  For i:=1 to k do
  begin
    x:=x0+Round(r*C[i]);
    y:=y0-Round(r*S[i]);
    Line(x0,y0,x,y);
    If m>1 then Snow(x,y,r div 3,m-1)
  end;
End;
Begin
  Driver:=Detect;
  InitGraph(Driver,Mode,'');
  For i:=1 to k do
  begin
    C[i]:=Cos(i*t);
    S[i]:=Sin(i*t);
  end;
  Snow(GetMaxX div 2,GetMaxY div 2,
       Round(160/(1-1/(Exp(n*Ln(3))))),n);
  ReadLn;
  CloseGraph;
End.
```

```
        begin
          x:=-0.15*x+0.28*y;
          y:=0.26*t+0.24*y+0.44;
        end
        else
          begin
            x:=0.0;
            y:=0.16*y;
          end;
      PutPixel(mid_x+Round(r*x),
               mid_y-Round(r*y),LightGreen);
  end;
  ReadLn;
  CloseGraph;
End.
```

5. Построение фрактальных изображений.

Фрактал – это самоподобный объект, у которого любая меньшая часть похожа на целый объект. Обычно фрактальные фигуры строятся с помощью рекурсивных подпрограмм. Простейшим примером фрактальной фигуры является снежинка:



Алгоритм рисования снежинки: из одной точки – центра вырастают k кристалликов-отрезков длины r, свободный конец каждого из которых служит центром новой снежинки с длиной кристаллика–отрезка, в 3 раза меньшей r. Указанный процесс продолжается n раз. Выше показаны снежинки при n=1,2 и k=6.

Для построения снежинок введем рекурсивную процедуру, параметрами которой будут координаты центра снежинок $x0$, $y0$, радиус–длина r и глубина рекурсии n. Для размещения рисунка в центре экран начальный радиус определим по формуле: $\frac{\dim\cdot(k-1)}{k^n-1}$, полагая dim=240, k=1/3. Получим: k=160/(1–1/3^n). Координаты концов отрезка

2-ой	*1*	*4*	*8*	*5*	*6*	*2*
	1	*2*	*8*	*5*	*6*	*4*
3-ий	*1*	*2*	*8*	*5*	*6*	*4*
	1	*2*	*4*	*5*	*6*	*8*
4-ый	*1*	*2*	*4*	*5*	*6*	*8*
	1	*2*	*4*	*5*	*6*	*8*
5-ый	*1*	*2*	*4*	*5*	*6*	*8*
	1	*2*	*4*	*5*	*6*	*8*

```
  For j:=i+1 to n do
    If A[j]<min then
      begin
        min:=A[j];
        i_min:=j;
      end;
  r:=A[i_min];
  A[i_min]:=A[i];
  A[i]:=r;
end;
```

Линейный выбор с подсчетом

Для сортировки элементов исходного вектора A размерности n, нужно сформировать вектор счетчиков S размерности n, каждый элемент которого будет показывать, в какой позиции должен стоять соответствующий элемент вектора A в упорядоченном списке.

На первом этапе в вектор счетчиков заносятся единицы. Для формирования вектора счетчиков нужно n-1 раз просмотреть элементы исходного вектора A.

При *первом* просмотре для первого элемента вектора A подсчитывается количество меньших элементов во всем векторе и это количество заносится в счетчик первого элемента, в счетчики элементов, больших первого, добавляются единицы.

При *втором* просмотре первый элемент вектора A исключается из рассмотрения и для второго элемента в оставшемся векторе подсчитывается количество меньших элементов (это количество заносится в счетчик второго элемента), в счетчики больших элементов добавляются единицы, и.т.д.

Перемещая элементы исходного вектора A в полученный вектор B в соответствии со значениями вектора счетчиков S, получим упорядоченный исходный вектор B.

Просмотр	Исходный вектор *A*	Вектор счетчиков S
1-ый	*2 4 8 5 6 1* Кол-во меньших эл-тов	*1 1 1 1 1 1* + *1 1 1 1 1* *2 2 2 2 2 1*
2-ой	*2 4 8 5 6 1*	*2 2 2 2 2 1* + *1 1 1 1* *2 3 3 3 3 1*

3-ий	*2 4 (8) 5 6 1*	*2 3 3 3 3 1* + *(3)* *2 3 6 3 3 1*
4-ый	*2 4 8 (5) 6 1*	*2 3 6 3 3 1* + *(1) 1* *2 3 6 4 4 1*
5-ый	*2 4 8 5 (6) 1*	*2 3 6 4 4 1* + *(1)* *2 3 6 4 5 1*

Таким образом, сформированный вектора счетчиков показывает, что первый элемент вектора *A* должен стоять в упорядоченном списке на втором месте, второй элемент вектора *A* – на третьем, третий элемент – на последнем и т.д.

```
For i:=1 to n do         {инициируем вектор счетчика}
  S[i]:=1;
For i:=1 to n-1 do       {формируем вектора счетчика}
begin
  k:=0;  {количество меньших элементов для i-го элемента}
  For j:=i+1 to n do
   If A[i]<A[j] then S[j]:=S[j]+1
{в счетчики больших элементов добавляем единицы}
                 else k:=k+1;
  S[i]:=S[i]+k;
end;
{формируем вектор B в соответствии со значениями вектора счетчика}
For i:=1 to n do
begin
   r:=S[i];
   B[r]:=A[i];
end;
```

Парный обмен

Метод парного обмена состоит из различного числа четных и нечетных просмотров.

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0{,}85 & -0{,}04 \\ -0{,}04 & 0{,}85 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 1{,}6 \end{bmatrix}, \text{ вероятность } p \in [0;0{,}85];$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0{,}16 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \text{ вероятность } p \in (0{,}99;1];$$

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} -0{,}15 & 0{,}28 \\ 0{,}26 & 0{,}24 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 0{,}44 \end{bmatrix}, \text{ вероятность } p \in (0{,}92;0{,}99].$$

```
Program Paporotnik;
  Uses Graph,Crt;
  Const
     iteration=500000;
  Var
   t,x,y,p                       :Real;
   k                             :LongInt;
   Driver,Mode,mid_x,mid_y,r :Integer;
Begin
  Driver:=Detect;
  InitGraph(Driver,Mode,'');
  mid_x:=GetMaxX div 2;
  mid_y:=GetMaxY;
  r:=Trunc(0.1*mid_y);
  Randomize;
  x:=1.0;
  y:=0.0;
  For k:=1 to iteration do
  begin
    p:=Random;
    t:=x;
    If p<=0.85 then
    begin
      x:=0.85*x+0.04*y;
      y:=-0.04*t+0.85*y+1.6;
    end
    else
      If p<=0.92 then
      begin
        x:=0.2*x-0.26*y;
        y:=0.23*t+0.22*y+1.6;
      end
      else
        If p<=0.99 then
```

```
  x:=50;
  rx:=30;                {ширина столбика}
  ras:=Round(500/N);     {расстояние между столбиками}
  osn:=250;              {координата основания}
  Rectangle(x,osn,500, osn+35);
  SetTextStyle(4,0,2);
  pr:=0;
  OutTextXY(100,30,
            'Знаменитые алмазы (масса в каратах)');
  max:=A[1];
  For i:=1 to N do
    If A[i]>max then max:=A[i];
  For i:=1 to N do
  begin
    OutTextXY(x+10,osn+5,Y[i]);
    h:=Trunc(A[i]*200/max);           {высота}
    SetFillStyle(9,i+10);
    Bar (x, 250-h,x+rx, osn);
    Str(a[i]:6:1,t);
    OutTextXY(x,250-h-12,t);
    x:=x+ras;
  end;
  ReadLn;
end;
Begin
  Driver:=Detect;
  InitGraph(Driver,Mode,'');
  Stol;
  Sect;
  CloseGraph;
End.
```

4. Построение вероятностных изображений.

Рассмотрим пример построения листа папоротника. Заданное множество точек строится с помощью четырех преобразовании координат точек плоскости, каждое из которых применяется с определенной вероятностью. Преобразования задаются матрицей коэффициентов и вектором смещения вдоль оси *y*:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 0{,}2 & -0{,}26 \\ 0{,}23 & 0{,}22 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ 1{,}6 \end{bmatrix}$$, вероятность $p \in (0{,}85;0{,}92]$;

При *четном* просмотре элемент, стоящий в четной позиции, сравнивается со следующим элементом, стоящим в нечетной позиции, и больший занимает нечетную позицию.

При *нечетном* просмотре элемент, стоящий в нечетной позиции, сравнивается со следующим элементом, стоящим в четной позиции, и больший занимает четную позицию.

За четный и нечетный просмотры подсчитывается количество перестановок. Просмотры прекращаются, когда за четный и нечетный просмотры не было сделано ни одной перестановки.

Кол-во перестановок	Нечетный просмотр	Четный просмотр
3	*2 4 8 5 6 1*	*2 4 5 8 1 6*
3	*2 4 5 1 8 6*	*2 4 1 5 6 8*
1	*2 1 4 5 6 8*	*1 2 4 5 6 8*
0	*1 2 4 5 6 8*	*1 2 4 5 6 8*

```
Repeat
  k:=0;                   {к-количество перестановок}
  For j:=1 to 2 do        {четный и нечетный просмотр}
  begin
    i:=j;
    While (i<n) do
    begin
      If A[i]>A[i+1] then
      begin
        r:=A[i];
        A[i]:=A[i+1];
        A[i+1]:=r;
        k:=k+1;
      end;
      i:=i+2;
    end;
  end;
Until k=0;
```

Метод стандартного обмена (пузырька)

Метод стандартного обмена при каждом просмотре вектора перемещает один элемент исходного вектора в соответствующую позицию, т.е. при первом просмотре наибольший элемент вектора перемещается в последнюю позицию, при втором просмотре элемент, следующий за наибольшим по величине, перемещается в предпоследнюю позицию и т.д. Для сортировки вектора нужно *n*-1 раз просмотреть элементы исходного вектора.

При *первом* просмотре первый элемент сравнивается со вторым и больший из них занимает вторую позицию, затем второй элемент сравнивается с третьим и больший занимает третью позицию и т.д. Когда *n*-1 элемент сравнивается с *n*-ым и больший занимает последнюю позицию, первый просмотр заканчивается.

Второй просмотр аналогичен первому с той лишь разницей, что последний элемент исключается из рассмотрения.

Каждый последующий просмотр исключает очередную установленную позицию из рассмотрения, тем самым укорачивая вектор.

Просмотр	Исходный вектор *A*
1-ый	*2 4 8 5 6 1* *2 4 5 8 6 1* *2 4 5 6 8 1* *2 4 5 6 1 8*
2-ой	*2 4 5 6 1 8* *2 4 5 1 6 8*
3-ий	*2 4 5 1 6 8* *2 4 1 5 6 8*
4-ый	*2 4 1 5 6 8* *2 1 4 5 6 8*
5-ый	*2 1 4 5 6 8* *1 2 4 5 6 8*

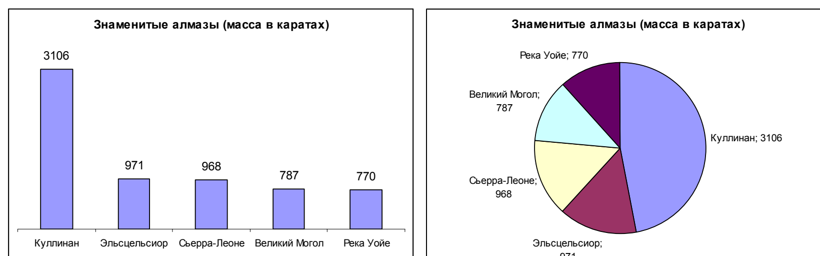
```
For i:=n-1 downto 1 do
For j:=1 to i do
If A[j]>A[j+1] then
begin
    r:=A[j];
    A[j]:=A[j+1];
    A[j+1]:=r;
end;
```



```
  Z                        :array[1..N] of Integer;
  T                        :String[15];
  max,sum                  :Real;
  y0,y1,y2,h,pr,x,rx,osn,
  ras,I                    :Integer;
{Круговая диаграмма}
Procedure Sect;
Begin
  OutTextXY(100,30,
          'Знаменитые алмазы (масса в каратах)');
{расчет общей суммы}
  sum:=0;
  For i:=1 to N do
    sum:=sum+A[i];
  For i:=1 to N do
    Z[i]:=Round(a[i]*360/sum);
  y0:=0;
{рисование сектора}
  For i:=1 to N do
  begin
     SetFillStyle(2,i+1);
     PiesLice(325,190,y0,y0+z[i],140);
     y0:=y0+z[i];
  end;
  y1:=50;
  y2:=70;
{рисование легенды и подписи к ней}
  For i:=1 to N do
  begin
    SetFillStyle(2,i+1);
    Bar3d(20,y1,60,y2,5,True);
    Str(A[i]:6:1,t);
    OutTextXY(70,y1+10,t);
    y1:=y1+30;
    y2:=y2+30;
  end;
  ReadLn;
End;
{Столбчатая диаграмма}
Procedure Stol;
Begin
  ClearDevice;
```

3. Построение круговых и столбчатых диаграмм.

Построить круговую и столбчатую диаграммы, содержащие сведения о знаменитых алмазах следующего вида:



Для построения диаграмм нужно определить 2 массива: первый содержит значения массы в каратах (массив A[1..5]), второй – названия алмазов (массив Y[1..5]).

Расчет высоты столбика производится следующим образом:

- находим самый большой алмаз (max). Пусть этому значению соответствует столбик в 200 пиксел;
- высота каждого столбика равна $h[i] = \frac{200 \cdot A[i]}{\max}$.

Градусная мера угла для построения круговых диаграмм рассчитывается так:

- находим сумму в каратах всех знаменитых алмазов (sum). Этому значению будет соответствует угол в 360^0;
- градусную меру каждого угла определим из формулы $z[i] = \frac{360 \cdot A[i]}{sum}$.

```
Program Demo_3;
Uses Graph,Crt;
Type
  Mas1=array[1..5] of Real;
  Mas2=array[1..5] of String[15];
Const
  N=5; {количество данных}
  A:Mas1=(3106,971,968,787,770); {значения}
  Y:Mas2=('Куллинан','Эльсценсиор',
          'Сьерра-Леоне','Великий Могол',
          'Река Уойе'); {подписи данных}
Var
  Driver,Mode            :Integer;
```


Метод просеивания

Метод просеивания работает точно также, как и метод стандартного обмена до тех пор, пока не нужно сделать перестановку. Пусть *k* – номер позиции переставляемого элемента. С позиции *k* начинается нисходящий просмотр элементов вектора в обратном порядке до тех пор, пока не нужно будет переставлять элементы вектора. Затем возобновляется восходящий просмотр элементов вектора с позиции *k*.

Таким образом, дойдя до последнего элемента вектора, получим упорядоченный список.

Исходный вектор *A*					
2	4	8	5	6	1
2	4	5	8	6	1
2	4	5	8	6	1
2	4	5	6	8	1
2	4	5	6	1	8
2	4	5	6	1	8
2	4	5	1	6	8
2	4	1	5	6	8
2	1	4	5	6	8
1	2	4	5	6	8

```
For i:=1 to n-1 do
If A[i]>A[i+1] then
begin
  r:=A[i];
  A[i]:=A[i+1];
  A[i+1]:=r;
  j:=i;
  While   (A[j]<A[j-1])   and
(j>1) do
  begin
    r:=A[j];
    A[j]:=A[j-1];
    A[j-1]:=r;
    j:=j-1;
  end;
end;
```

Метод линейной вставки

При сортировке исходного вектора *A* методом линейной вставки последовательно просматриваются элементы вектора *A* и формируется упорядоченный вектор *B*. До начала сортировки считается, что длина упорядоченного вектора равно 0. Как только длина упорядоченного вектора станет равной длине исходного вектора, сортировка закончена.

Первый элемент вектора *A* помещается в первую позицию вектора *B*. Длина вектора становится равной 1.

Второй элемент вектора *A* сравнивается со всеми элементами вектора *B*. Если он больше всех элементов вектора *B*, то помещаем его в конец вектора и при этом длина вектора *B* увеличивается на единицу. Если в векторе *B* найдется элемент, меньший сравниваемого, то фиксируем позицию этого элемента, и все элементы вектора *B*, начиная с этой позиции, перемещаем на одну позицию вправо, начиная с последнего. На освободившееся место помещаем сравниваемый элемент. Длину вектора *B* увеличиваем на единицу.

В дальнейшем, последовательно просматривая элементы вектора **А**, сравниваем их с элементами вектора *B*, начиная с первого, до тех пор, пока не встретиться больший. Этот больший элемент и все последующие элементы вектора *B* передвигаются на одну позицию вправо. Таким образом освобождается место, на которое вставляется новый элемент.

Просмотр	Исходный вектор А	Полученный вектор В
1-ый	*(2) 4 8 5 6 1*	*(2)*
2-ой	*2 (4) 8 5 6 1*	*2 (4)*
3-ий	*2 4 (8) 5 6 1*	*2 4 (8)*
4-ый	*2 4 8 (5) 6 1*	*2 4 8→* *2 4 (5) 8*
5-ый	*2 4 8 5 (6) 1*	*2 4 5 8→* *2 4 5 (6) 8*
6-ой	*2 4 8 5 6 (1)*	*2 4 5 6 8→* *2 4 5 6→ 8* *2 4 5→ 6 8* *2 4→ 5 6 8* *2→ 4 5 6 8* *2 4 5 6 8* *(1) 2 4 5 6 8*

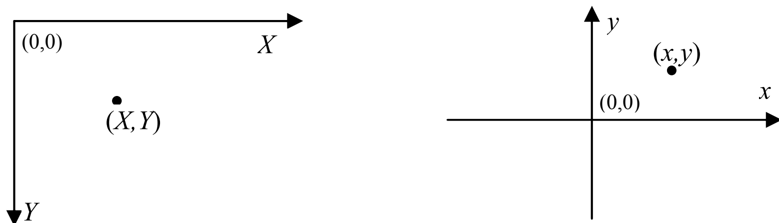
```
B[1]:=A[1];
k:=1;
For i:=2 to n do
begin
  j:=1;
  While (j<=k) and (a[i]>b[j]) do
    j:=j+1;
  If j=k+1 then
           begin
             b[j]:=a[i];
```



```
  {Рисуем оси}
  SetColor(Red);
  SetLineStyle (SolidLn,0,ThickWidth);
  Line(0,wy,GetMaxX,wy); {ox}
  Line(wx,0,wx,GetMaxY); {oy}
  {Подписи осей}
  SetColor(White);
  OutTextXY(wx+7,4,'Y');
  OutTextXY(GetMaxX-8,wy+6,'X');
  {Вертикальные линии сетки}
  SetColor(Green);
  SetLineStyle (SolidLn,0,NormWidth);
  i:=1;
  While (wx-i*wm>0) or (wx+i*wm<GetMaxX) do
  begin
    Line(wx-i*wm,0,wx-i*wm,GetMaxY);
    Line(wx+i*wm,0,wx+i*wm,GetMaxY);
    Inc(i);
  end;
  {Горизонтальные линии сетки}
  i:=1;
  While (wy-i*wn>0) or (wy+i*wn<GetMaxY) do
  begin
    Line(0,wy-i*wn,GetMaxX,wy-i*wn);
    Line(0,wy+i*wn,GetMaxX,wy+i*wn);
    Inc(i);
  end;
  {Строим график точками}
  i:=a*wm;
  While i<=c*wm do
  begin
    x:=i/wm;
    y:=sin(x);
    PutPixel(Round(wx+wm*x),Round(wy+wn*y),White);
    i:=i+1;
  end;
  ReadLn;
  CloseGraph;
End.
```

2. Построение графика функции.

Для построения графика функции прежде всего нужно построить систему координат, так как для графического режима она отличается от привычной для пользователя (ось *oy* направлена сверху вниз, начало координат находится в левом верхнем углу экрана).



Система координат графического экрана | Декартова система координат

Для преобразования координат точки (x,y) декартовой системы координат в координаты точки (X,Y) системы координат графического режима можно использовать преобразования:

$$\begin{cases} X = w_x + x \cdot w_m \\ Y = w_y + y \cdot w_n \end{cases}$$

где w_m, w_n – масштабные коэффициенты по осям, (w_x,w_n) – координаты центра начала декартовой системы координат на графическом экране.

Пример: построить график функции $y=\sin(x)$.

```
Program Demo_2;
Uses Graph;
Const
  a=-10;    {[a,c] – отрезок, на котором строится график функции}
  c=10;
  wm=50;    {масштабный множитель по оси ox}
  wn=50;    {масштабный множитель по оси oy}
Var
  Driver,Mode:Integer;
  i,wx,wy    :Integer;
  x,y:Real;
Begin
  Driver:=Detect;
  InitGraph(Driver,Mode,'');
  {Задаем координаты начала системы координат в центре экрана}
  wx:=GetMaxX div 2;
  wy:=GetMaxY div 2;
```



```
      k:=k+1;
    end
    else
    begin
      For l:=k+1 downto j+1 do
        b[l]:=b[l-1];
      b[j]:=a[i];
      k:=k+1;
    end;
end;
```

Сравнение методов по основным характеристикам
для вектора размерности n

Метод сортировки	Минимальное и максимальное число сравнение	Количество перемещений или обменов
Линейный выбор	n^2, n^2	n перемещений
Линейный выбор с обменом	$\frac{n^2}{2}$, $\frac{n^2}{2}$	n обменов
Линейны выбор с подсчетом	$\frac{n^2}{2}$, $\frac{n^2}{2}$	n перемещений и $\frac{n^2}{2}$ подсчетов
Парный обмен	n, $\frac{n^2}{2}$	$\frac{n^2}{2}$ обменов
Метод стандартного обмена	n, $\frac{n^2}{2}$	$\frac{n^2}{2}$ обменов
Метод просеивания	n, $\frac{n^2}{2}$	$\frac{n^2}{2}$ обменов
Метод линейной вставки	n, $\frac{n^2}{2}$	$\frac{n^2}{2}$ перемещений

МНОЖЕСТВА

Множество – это структурированный тип данных, представляющий набор взаимосвязанных по какому-либо признаку или группе признаков объектов, которые можно рассматривать как единое целое. Элементы множества должны принадлежать к одному из скалярных типов, кроме вещественного и типов `Word`, `Integer`, `LongInt`. Количество элементов множества не должно превышать 256.

В выражениях на языке Pascal элементы множества указываются в квадратных скобках, например:

```
[1,2,3,4],  ['a','b','c'],   ['a'..'z']
```

Если множество не имеет элементов, оно называется *пустым* и обозначается [].

Форматы описания множества:

1–ый способ:

```
Type
  имя_типа=set of тип_элементов;
Var
  идентификатор: имя_типа;
```

2–ой способ:

```
Var
  идентификатор: set of тип_элементов;
```

3-ий способ:

```
Type
  имя_типа=set of тип_элементов;
Const
  идентификатор:имя_типа=значения;
```

Примеры описания множеств:

1–ый способ:

```
Type
  days=set of 1..31;
Var
  p: days;
```

2–ой способ:

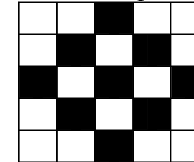
```
Var
  ch: set of char;
  mn: set of byte;
```

3-ий способ:

```
Type
  days=set of 1..31;
```


рицу, содержащую вид этого изображения. Каждый элемент матрицы с координатами (i,j) – номер цвета, которым нужно закрасить прямоугольник, находящийся на пересечении i-ой строки и j-го столбца.

Пример. Создать мозаичное изображение вида:



```
Program Demo_1;
Uses Crt,Graph;
Const
  h=10;  {высота прямоугольника}
  B: array[1..5,1..5] of 0..2=
   ((0,0,1,0,0),  {задаем матрицу размером 5x5}
    (0,1,0,1,0),  {тип 0..2 – количество используемых цветов}
    (1,0,2,0,1),
    (0,1,0,1,0),
    (0,0,1,0,0));
Var
   Driver,Mode,i,j,x,y:Integer;
Begin
  Driver:=Detect;
  InitGraph(Driver,Mode,'');
  For i:=1 to 5 do
  begin
    y:=i*h;
    for j:=1 to 5 do
    begin
      x:=75+j*h;
      case B[i,j] of
      0:SetFillStyle(1,White);  {меняем цвет заполнения}
      1:SetFillStyle(1,Blue);  {в зависимости от значения}
      2:SetFillStyle(1,Yellow)  {элемента матрицы}
      end;
      Bar(x,y,x+h,y+h);  {рисуем прямоугольник}
    end;
  end;
  ReadLn;
  CloseGraph;
End.
```

```
  ReadLn;
  CloseGraph;
End.
```

3-ий способ.

```
Program Demo_3;
Uses Graph,Crt;
Var
  P    :Pointer;  {определяем нетипизированный указатель}
  Size :Word;
  Driver,Mode,i:Integer;
Begin
  Driver:=Detect;
  InitGraph(Driver,Mode,'');
  {рисуем закрашенный шарик}
  SetBkColor(Black);
  SetColor(Red);
  Circle(30,30,15);
  SetFillStyle(SolidFill,Blue);
  FloodFill(30,30,Red);
  size:=ImageSize(5,5,50,50);  {определяет объем памяти,}
                               {необходимый для сохранения изображения}
  GetMem(P,Size);  {выделяет память в динамически}
                   {распределяемой области размером в size байт}
  GetImage(5,5,50,50,P^);  {запоминаем изображение}
  PutImage(i,i,P^,XorPut);  {стираем изображение}
  for i:=5 to 300 do
  begin
    PutImage(i,i,P^, XorPut);  {выводим изображение}
    Delay(3000);               {пауза}
    PutImage(i,i,P^,XorPut);   {стираем изображение}
  end;
  ReadLn;
  CloseGraph;
End.
```

Примеры программ работы с графикой

1. Построение мозаичных изображений.

Мозаичное изображение можно построить с помощью рисования прямоугольников. Для задания изображения нужно сформировать мат-


```
  digc=set of '0'..'9';
Const
  WorkDays:days=[1..5,8..12,15..19,22..26,29,30];
  EvenDigits:digc=['0','2','4','6','8'];
  Flag: set of char=[];
  English:set of char=['a'..'z'];
```

Операции над множествами

= проверка равенства множеств. Два множества считаются равными, если они состоят из одних и тех же элементов. Порядок следования элементов значения не имеет.

<> не равно. Два множества считаются не равными, если они отличаются по количеству элементов или по значению хотя бы одного элемента:

A	*B*	Операция	Результат
[1,2,3]	[1,2,3,4]	A<>B	True
['a'..'z']	['b'..'z']	A<>B	True

>= используется для определения принадлежности множеств. Результат *A*>=*B* равен True, если все элементы множества B содержатся во множестве A:

A	*B*	Операция	Результат
[1,2,3,4]	[2,3,4]	A>=B	True

<= аналогична >=

+ объединение множеств. Объединением двух множеств является третье множество, содержащее элементы обоих множеств:

A	*B*	Операция	Результат
[1,2,3]	[2,3,4]	A+B	[1,2,3,4]

* пересечение множеств. Пересечением двух множеств является третье множество, входящие одновременно в оба множества:

A	*B*	Операция	Результат
[1,2,3]	[2,3,4]	A*B	[2,3]

– разность множеств. Разностью двух множеств является третье множество, которое содержит элементы первого множества, не входящие во второе множество:

A	*B*	Операция	Результат
[1,2,3,4]	[1,3,4]	A–B	[2]

In – проверка принадлежности какого-либо значения указанному множеству. Обычно применяется в условных операторах:

```
Program Mnoj;
Const
  Simple:set of Byte=[2,3,5,7,11,13,17,19];
Var
  a: Byte;
Begin
  ReadLn(a);
  If a In Simple then WriteLn ('Число простое');
End.
```

Процедуры работы с множествами

Incude(множество, элемент) – включает новый элемент во множество.

Exclude(множество, элемент) – исключает элемент из множества.

Замечание: добавлять элементы во множество можно и без использования процедуры Include следующим образом:

```
Program DemoInc;
  Var
    S:set of Char;
    ch:Char;
Begin
  S:=[];          {определили пустое множество}
  ReadLn (ch);    {ввели значение}
  S:=S+[ch];      {добавили во множество}
End.
```

Пример. Дана строка. Сформировать и распечатать множество гласных букв английского алфавита, содержащихся в строке.

```
Program DemoSet;
  Const
    Engl_Gl:set of Char=['a','e','u','i','o','y'];
```



```
    FloodFill(i,i,Red);  {закрасили}
    SetColor(Black);  {цвет границы – черный}
    Circle(i,i,15);  {нарисовали черный контур
                       на черном фоне, шарик исчез}
    Delay(3000);  {пауза}
  end;
  {чтобы шарик остался на экране, нарисуем его еще раз}
  SetColor(Red);
  Circle(i,i,15);
  SetFillStyle(SolidFill,Blue);
  FloodFill(i,i,Red);
  ReadLn;
  CloseGraph;
End.
```

2-ой способ.

```
Program Demo_2;
Uses Graph,Crt;
Var
  P    :Pointer;  {определяем нетипизированный указатель}
  Size :Word;
  Driver,Mode,i:Integer;
Begin
  Driver:=Detect;
  InitGraph(Driver,Mode,'');
  {рисуем закрашенный шарик}
  SetBkColor(Black);
  SetColor(Red);
  Circle(30,30,15);
  SetFillStyle(SolidFill,Blue);
  FloodFill(30,30,Red);
  size:=ImageSize(5,5,50,50);  {определяет объем памяти,}
                    {необходимый для сохранения изображения}
  GetMem(P,Size);  {выделяет память в динамически}
                  {распределяемой области размером в size байт}
  GetImage(5,5,50,50,P^);  {запоминаем изображение}
  for i:=5 to 300 do
  begin
    PutImage(i,i,P^,NormalPut);  {выводим изображение}
    Delay(3000);                 {пауза}
  end;
```

или в двоичном виде 1111) эта операция даст код 0000=0=`Black`, для `Red` – 4=0100 получим 1011=11=`LightCyan` и т.д. Операция `XorPut`, примененная к тому же месту экрана, откуда была получена копия, сотрет эту часть экрана. Если эту операцию применить дважды к одному и тому же участку, вид изображения на экране не изменится. Таким способом можно перемещать изображения по экрану, создавая иллюзию движения.

Способы создания движущихся изображений

1-ый способ. Дважды рисуя одно и то же изображение: первый раз цветом, отличным от цвета фона, второй раз – цветом фона.

2-ой способ. Используя процедуры и функции работы с видеопамятью. Изображение движется с помощью операции `NormalPut`.

3-ий способ. Используя процедуры и функции работы с видеопамятью. Изображение движется с помощью операции `XorPut`.

Пример программы движения шарика, падающего с левого верхнего угла экрана по диагонали в правый нижний.

1-ый способ.

```
Program Demo_1;
Uses
  Graph,Crt;
Var
  Driver,Mode,i:Integer;
Begin
  Driver:=Detect;
  InitGraph(Driver,Mode,'');
  SetBkColor(Black);
  For i:=25 to 300 do {схема движения шарика}
  begin
  {рисуем шарик}
    SetColor(Red);      {цвет границы – красный}
    Circle(i,i,15);     {рисуем контур шарика}
    SetFillStyle(SolidFill,Blue);
    {стиль заполнения – синий фон}
    FloodFill(i,i,Red); {закрасили}
    {стираем шарик, закрашивая его цветом фона}
    SetFillStyle(SolidFill,Black);
    {стиль заполнения – черный фон}
```



```
  Var
    St  :String;
    Gl  :set of char;
    i   :Integer;
    ch  :Char;
Begin
  WriteLn('Введите строку:');
  ReadLn(St);
  {Формируем множество}
  Gl:=[];
  For i:=1 to Length(St) do
    If St[i] In Engl_Gl then Include(Gl,St[i]);
  {Вывод элементов множества}
  WriteLn ('Множество гласных букв, содержащихся в
           строке:');
  For ch:='a' then 'z' do
    If ch In Gl then Write (ch:2);
End.
```

ЗАПИСИ

Фиксированные записи

Запись – это структурированный тип данных, состоящий из фиксированного числа компонент разного типа.

Определение типа записи начинается идентификатором `record` и заканчивается зарезервированным словом `end`. Между ними заключен список компонентов, называемых *полями*, с указанием идентификаторов полей и типа каждого поля.

Формат описания записи:

```
Type
    имя_типа=record
                идентификатор_поля: тип_компоненты;
                    …
                идентификатор_поля: тип_компоненты
            end;
Var
    идентификатор: имя_типа;
```

Пример:

```
Type
  Sved=record
        Fam     :String[20];
        God     :Integer;
        Nom_gr  :String[10];
        Adres:String[50]
     end;
Var
    Ank,S:Sved;
```

Обращение к полям записи осуществляется с помощью идентификатора переменной и идентификатора поля, разделенных точкой. Такая комбинация называется составным именем.

```
Ank.Fam
```

Все действия производятся над полями записи, только операцию присваивания можно выполнить над всей записью.

```
S:=Ank;
```

Массив записей описывается следующим образом:

```
Var
    Mas:array[1..10] of Sved;
```


`ImageSize (x1,y1,x2,y2)` – функция возвращает размер памяти в байтах, необходимый для размещения прямоугольного фрагмента изображения. Здесь (x1,y1) – координаты левого верхнего, (x2,y2) – правого нижнего углов фрагмента изображения.

`GetImage(x1,y1,x2,y2,Buf)` – процедура помещает в память копию прямоугольного фрагмента изображения. Здесь (x1,y1) – координаты левого верхнего, (x2,y2) – правого нижнего углов фрагмента изображения; `Buf` — переменная или участок кучи, куда будет помещена копия видеопамяти с фрагментом изображения. Размер `Buf` должен быть получен с помощью функции `ImageSize (x1,y1,x2,y2)`.

`PutImage(x,y,Buf,Mode)` – процедура выводит в заданное место экрана копию фрагмента изображения, ранее помещенную в память процедурой `GetImage`. Здесь (x,y) – координаты левого верхнего угла того места на экране, куда будет скопирован фрагмент изображения; `Buf` – переменная или участок кучи, откуда берется изображение; `Mode` – способ копирования. Параметр *Mode* определяет способ взаимодействия вновь размещаемой копии с уже имеющимся на экране изображением. Взаимодействие осуществляется путем применения кодируемых этим параметром логических операций к каждому биту копии и изображения. Для указания применяемой логической операции можно использовать одну из следующих предварительно определенных констант:

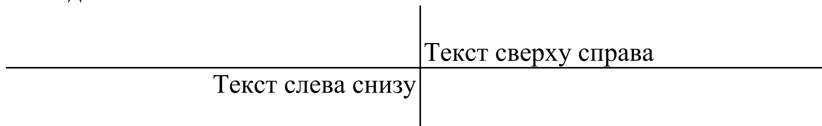
```
Const
  NormalPut=0; {Замена существующего изображения на копию}
  XorPut=1;    {Исключительное ИЛИ}
  OrPut=2;     {Объединительное ИЛИ}
  AndPut=3;    {Логическое И}
  NotPut=4;    {Инверсия изображения}
```

Наиболее часто используются операции `NormalPut`, `XorPut` и `NotPut`. Первая из них просто стирает часть экрана и на это место помещает копию из памяти в том виде, как она там сохраняется. Операция `NotPut` делает то же самое, но копия выводится в инверсном виде. Для монохромного режима это означает замену светящихся пиксел на темные и наоборот. В цветном режиме операция `NotPut` применяется к коду цвета каждого пиксела. Например, для `White` (код 15

```
    Font      :Word; {Номер шрифта}
    Direction:Word; {Направление}
    CharSize :Word; {Код размера}
    Horiz     :Word; {Горизонтальное выравнивание}
    Vert      :Word  {Вертикальное выравнивание)
  end;
```

InstallUserFont (*имя_файла*) – функция позволяет программе использовать нестандартный векторный шрифт, который находится в указанном файле. Файл должен располагаться в текущем каталоге. Функция возвращает идентификационный номер нестандартного шрифта, который может использоваться при обращении к процедуре SetTextStyle.

Пример работы со шрифтами и выравниванием текста следующего вида:




```
Uses Crt,Graph;
Var
  Driver, Mode:Integer;
Begin
  Driver:=Detect;
  InitGraph(Driver,Mode,'');
  {Выводим перекрестные линии в центре экрана}
  Line(0,GetMaxY div 2, GetMaxX,GetMaxY div 2);
  Line(GetMaxX div 2,0,GetMaxX div 2,GetMaxY);
  {Располагаем текст справа сверху от центра}
  SetTextStyle(TriplexFont,HorizDir,3);
  SetTextJustify(LeftText,BottomText);
  OutTextXY(GetMaxX div 2,GetMaxY div 2,
            'Текст справа сверху');
  {Располагаем текст слева и снизу от центра}
  SetTextJustify(RightText,TopText);
  OutTextXY(GetMaxX div 2,GetMaxY div 2,
            'Текст слева снизу');
  Readln;
  CloseGraph;
End.
```


Для более простого обращения к полям записей используется оператор присоединения with:

With *переменная_типа_запись* **do**
оператор;

Один раз указав переменную типа запись в операторе with, можно работать с именами полей, как с обычными переменными, т.е. без указания перед идентификатором поля имени переменной, определяющей запись:

Использование переменной типа запись

```
With Ank do
begin
  Write(Fam);
  ...
end;
```

Использование массива записей

```
For i:=1 to N do
With Mas[i] do
begin
  Write(Fam);
  ...
end;
```

Сравните, без использования оператора with:

Использование переменной типа запись

```
  Write(Ank.Fam);
  ...
```

Использование массива записей

```
For i:=1 to N do
begin
  Write(Mas[i].Fam);
  ...
end;
```

Паскаль допускает вложение записей друг в друга (т.е. поле записи может быть в свою очередь тоже записью), соответственно оператор with может быть вложенным. Уровень вложения не должен превышать 9.

```
With rv1 do
  With rv2 do
    …
      With rvn do оператор;
```

Вместо последней записи можно использовать более короткую:
With rv1,rv2,…,rvn **do** оператор;
Здесь rv1,rv2,…,rvn переменные типа запись.

Пример вложенных записей:

```
Type
  Tfio=record
         Fam      :String;
         Name :String;
         Otch :String
```

```
    end;
  Tdata=record
          Month   :1..12;
          Year:1900..2005;
          Day     :1..31;
        end;
  Sved=record
         Fio:Tfio;
         Data:Tdata;
         Adr:String
       end;
```

Записи можно описывать с помощью типизированных констант следующим образом:

Const

идентификатор:*тип*=(*список_значений_полей*);

Список значений полей представляет собой список из последовательностей вида:

имя_поля: *константа*

Элементы списка отделяются друг от друга двоеточиями.

Пример:

```
Type
  point=record
    x,y:real
  end;
  vect=array[0..1] of point;
  month=(Jan, Feb, Mar, Apr, May, Jun, Jly, Aug,
         Sep, Oct, Nov, Dec);
  date=record
         d:1..31;
         m:month;
         y:1900..2005
       end;
Const
  Origon:point=(x:0; y:-1);
  Line:vector=((x:-3.1; y:1.5),
               (x:5.9; y:3.0));
  SomeDay:date=(d:16; m:Mar; y:2005);
```

Пример. Описать массив записей, содержащих сведения о сдаче студентами сессии. Структура записи: фамилии студента, номер группы, результаты сдачи трех экзаменов. Распечатать список студентов,

SetTextJustify(*горизонтальное_выравнивание*, *вертикальное_выравнивание*) – процедура задает выравнивание выводимого текста по отношению к текущему положению указателя или к заданным координатам. Выравнивание определяет, как будет размещаться текст – левее или правее указанного места, выше, ниже или по центру. Здесь можно использовать константы:

```
Const
  LeftText=0;     {Указатель слева от текста}
  CenterText=1;   {Симметрично слева и справа, сверху и снизу}
  RightText=2;    {Указатель справа от текста}
  BottomText=0;   {Указатель снизу от текста}
  TopText=2;      {Указатель сверху от текста}
```

SetUserCharSize(x1,x2,y1,y2) – процедура изменяет размер выводимых символов в соответствии с заданными пропорциями. Здесь x1,x2,y1,y2 – выражения типа Word, определяющие пропорции по горизонтали и вертикали. Процедура применяется только по отношению к векторным шрифтам. Пропорции задают масштабный коэффициент, показывающий во сколько раз увеличится ширина и высота выводимых символов по отношению к стандартно заданным значениям. Коэффициент по горизонтали находится как отношение x1 к x2, по вертикали – как отношение y1 к y2. Чтобы, например, удвоить ширину символов, необходимо задать x1=2 и x2=1. Стандартный размер символов устанавливается процедурой SetTextStyle, которая отменяет предшествующее ей обращение к SetUserCharSize.

TextWidth (*текст*) – функция возвращает длину в пикселах выводимой текстовой строки. Учитываются текущий стиль вывода и коэффициенты изменения размеров символов, заданные соответственно процедурами SetTextStyle и SetUserCharSize.

TextHeight (*текст*) – функция возвращает высоту шрифта в пикселах.

GetTextSettings (TextInfo) – процедура возвращает текущий стиль и выравнивание текста. Здесь TextInfo – переменная типа TextSettingsType, который в модуле Graph определен следующим образом:

```
Type
  TextSettingsType=record
```

Вывод текста

`OutText (`*текст*`)` – процедура выводит текстовую строку, начиная с текущего положения указателя.

`OutTextXY (x,y,`*текст*`)` – процедура выводит строку, начиная с позиции (x,y).

`SetTextStyle(`*шрифт*`,`*направление*`,`*размер*`)` – процедура устанавливает стиль текстового вывода на графический экран. Здесь *шрифт* – код (номер) шрифта; *направление* – код направления; *размер* – код размера шрифта.

Код шрифта задается одной из следующих предварительно определенных констант:

```
Const
  DefaultFont=0;       {Точечный шрифт 8x8}
  TriplexFont=1;       {Утроенный шрифт TRIP.CHP}
  SmallFont=2;         {Уменьшенный шрифт LITT.CHR}
  SansSerifFont=3;     {Прямой шрифт SANS.CHR)
  GothicFont=4;        {Готический шрифт GOTH.CHR}
```

Шрифт *DefaultFont* входит в модуль *Graph* и доступен в любой момент. Это – единственный матричный шрифт, т.е. его символы создаются из матриц 8x8 пиксел. Все остальные шрифты – векторные: их элементы формируются как совокупность векторов (штрихов), характеризующихся направлением и размером. Векторные шрифты отличаются более богатыми изобразительными возможностями, но главная их особенность заключается в легкости изменения размеров без существенного ухудшения качества изображения. Каждый из этих шрифтов размещается в отдельном дисковом файле. Если нужно использовать какой-либо векторный шрифт, соответствующий файл должен находиться в текущем каталоге, в противном случае вызов этого шрифта игнорируется и подключается стандартный шрифт.

Для задания *направления* выдачи текста можно использовать константы:

```
Const
  HorizDir=0;  {Слева направо}
  VertDir=1;   {Снизу вверх}
```

Каждый шрифт способен десятикратно изменять свои размеры. *Размер* выводимых символов может иметь значение в диапазоне от 1 до 10 (точечный шрифт – в диапазоне от 1 до 32). Если значение параметра равно 0, устанавливается размер 1, если больше 10 – размер 10.

получающих стипендию. Условие получения стипендии – средний балл больше 5.

```
Program Zapisi;
  Uses Crt;
  Type Sved=record
         Fio:String[50];  {Фамилия}
         Nom:String[10];  {Номер группы}
         b1,b2,b3:0..10;  {Результаты сдачи экзаменов}
         sb:Real;         {Средний балл}
            end;
  Var
    Mas:array[1..10] of Sved;
    i,N:Byte;
{формирование массива записей}
Procedure Vvod;
Begin
  Write ('Введите количество записей N=');
  ReadLn (N);
  For i:=1 to N do
  With Mas[i] do
  begin
    Clrscr;
    Write ('ФИО -->');
    ReadLn(Fio);
    Write ('Группа -->');
    ReadLn(Nom);
    Write ('Оценки -->');
    ReadLn(b1,b2,b3);
    sb:=(b1+b2+b3)/3;
  end
end;
{вывод исходного массива записей}
Procedure Vivod;
begin
  clrscr;
  WriteLn ('Сведения о студентах:');
  WriteLn ('Фамилия  Группа Оценки  Средний балл');
  For i:=1 to N do
  With Mas[i] do
    WriteLn (Fio:10,Nom:10,b1:2,b2:2,b3:2,sb:5:1);
  Repeat Until KeyPressed;
```

```
end;
{вывод списка студентов, получающих стипендию}
Procedure Obr;
begin
  Clrscr;
  WriteLn ('Студенты, получающие стипендию');
  WriteLn ('Фамилия  Группа Оценки  Средний балл');
  For i:=1 to N do
  With Mas[i] do
   If sb>5 then
    WriteLn (Fio:10,Nom:10,b1:2,b2:2,b3:2,sb:5:1);
  Repeat Until KeyPressed;
end;
Begin
  Vvod;
  Vivod;
  Obr;
End.
```

Записи с вариантами

Рассмотренные выше записи имеют строки определенную структуру. В языке Pascal имеется возможность задать тип записи, содержащий произвольное число вариантов структуры. Такие записи называются записями с вариантами. Записи с вариантами обеспечивают средства объединения записей, которые похожи, но не идентичны по форме. Они состоят из необязательной фиксированной и вариантной частей. Использование фиксированной части не отличается от описанного выше. Вариантная часть формируется с помощью оператора `case`. Он задает особое поле записи – поле признака, которое определяет, какой из вариантов в данный момент будет активизирован. Значением признака в каждый текущий момент выполнения программы должна быть одна из расположенных далее констант. Константа, служащая признаком, задает вариант записи и называется константой выбора.

Формат описания записи с вариантами:

Type

имя_типа=**record**

{фиксированная часть}

идентификатор_поля: *тип_компоненты*;

…

идентификатор_поля: *тип_компоненты*;


```
          (x:150;y:100),
          (x:200;y:50));
Var
  Driver, Mode:Integer;
Begin
  Driver:=Detect;
  InitGraph(Driver,Mode,'');
  {стиль линии}
  SetLineStyle(SolidLn,0,
            ThickWidth);
  {стиль заполнения, синий фон}
  SetFillStyle(SolidFill,Blue);
  {цвет границы – красный}
  SetColor(Red);
  FillPoly(7,mas);
  ReadLn;
  CloseGraph;
End.
```

`FillEllipse (x,y,`*x_радиус*`,`*y_радиус*`)` – процедура обводит линией и заполняет эллипс. Здесь (x,y) – координаты центра, *x_радиус*`,`*y_радиус* – горизонтальный и вертикальный радиусы эллипса в пикселах. Эллипс обводится линией, заданной процедурами `SetLineStyle` и `SetColor`, и заполняется с использованием параметров, установленных процедурой `SetFillStyle`.

`Sector (x,y,` *начальный_угол*`,` *конечный_угол*`,` *x_радиус*`,` *y_радиус*`)` – вычерчивает и заполняет эллипсный сектор с центром в точке (x,y) от *начального* до *конечного угла* заданными *x_радиусом* по горизонтали и *y-радиусом* по вертикали.

`PieSlice (x,y,`*начальный_угол*`,`*конечный_угол*`,`*радиус*`)` – вычерчивает и заполняет сектор окружности с центром в точке (x,y) от *начального* до *конечного угла* заданным *радиусом*. Сектор обводится линией, заданной процедурами `SetLineStyle` и `SetColor` и заполняется с помощью параметров, определенных процедурой `SetFillStyle`. Процедуру удобно использовать при построении круговых диаграмм.

качестве значение этого параметра может использоваться одна из следующих определенных в модуле `Graph` констант:

```
Const
  TopOn =True;
  TopOff=False;
```

При вычерчивании используется текущий стиль линий (`SetLineStyle`) и текущий цвет (`SetColor`). Передняя грань заливается текущим стилем заполнения (`SetFillStyle`).

Процедура обычно применяется для построения столбиковых диаграмм. Следует учесть, что параллелепипед «прозрачен», т.е. за его незакрашенными гранями могут быть видны другие элементы изображения.

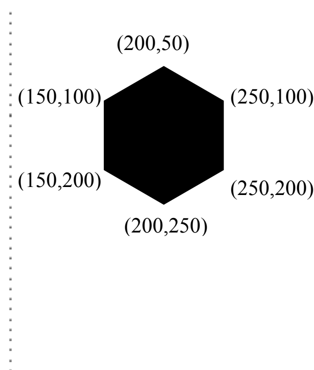
`FillPoly (n,Coords)` – процедура обводит линией и закрашивает замкнутый многоугольник. Здесь `n` – количество вершин замкнутого многоугольника, `Coords` – переменная типа `PoinType`, содержащая координаты вершин. Координаты вершин задаются парой значений типа `Integer`: первое определяет горизонтальную, второе – вертикальную координаты. Для них можно использовать следующий определенный в модуле тип:

```
Type
  PointType=record
    x,y:Integer
  end;
```

Стиль и цвет линий контура задаются процедурами `SetLineStyle` и `SetColor`, тип и цвет заливки – процедурой `SetFillStyle`.

Пример использования процедуры FillPoly. Рисуем и закрашиваем замкнутый многоугольник.

```
Uses
  Graph,Crt;
Type Tmas=array [1..7]
          of PointType;
{определяем массив записей
 вершин фигуры}
Const mas:Tmas=
         ((x:200;y:50),
          (x:250;y:100),
          (x:250;y:200),
          (x:200;y:250),
          (x:150;y:200),
```




```
{вариантная часть}
Case поле_признака: имя_типа of
константа_выбора_1: (поле,…:тип);
…
константа_выбора_n: (поле,…:тип)
end;
```

У части `case` нет отдельного `end`. Одно слово `end` заканчивает всю конструкцию записи с вариантами.

Пример:

```
Type
  tzap=record
  {фиксированная часть}
  Nomer   :Byte;
  Article :String;
  {вариантная часть}
  Case flag: Boolean of
    True: (cena1:Integer);
    False: (cena2:Integer);
  end;
Var
  zap:tzap;
```

При использовании записей с вариантами следует придерживаться следующих правил:

- все имена полей должны отличаться друг от друга по крайней мере одним символом, даже если они встречаются в разных вариантах;
- запись может иметь только одну вариантную часть, причем, она должна размещаться в конце записи;
- если поле, соответствующее какой-либо метке, является пустым, то оно записывается следующим образом:
  метка: ();

Записи с вариантами также можно задавать с помощью типизированных констант. При задании такой записи указываются только один из возможных вариантов констант:

```
Type
  Forma=record
  Case Boolean of
  True :(BirthPlace:String[40]);
  False:(Country:String[20]);
```

```
        EntryPort:String[20];
        EntryDate:array[1..3] of Word;
        Count:Word)
  end;
Const
  Person1:forma=(Country:'Норвегия';
                 EntryPort:'Мурманск';
                 EntryDate:(16, 3, 89);
                 Count: 12);
  Person2:(BirthPlace:'Москва');
```

Пример записей с вариантами. Ученики некоторого класса подразделяются на 3 группы:

- занимающиеся в спортивном кружке,
- занимающиеся в кружке туризма,
- нигде не занимающиеся.

В информации о школьниках первой группы нужно указать:

- вид спорта (легкая атлетика, футбол, волейбол, баскетбол),
- спортивный разряд (первый, второй, третий, кандидат в мастера спорта, мастер спорта);

второй группы:

- вид туризма (водный, пеший, горный),
- категорию самого сложного похода (1-6);

третьей группы:

- причину неучастия (болен, не успевает).

Описать массив записей с вариантами, имеющий указанную структуру. Организовать ввод и вывод массива.

```
Program School;
Uses Crt;
Type
  Tzan=(Sport,Turizm,Bez);  {вид занятий}
  Tvid=(L_a,Fut,Vol,Bas);   {вид спорта}
  Trazr=(r1,r2,r3,kms,ms);  {разряд}
  Tvidt=(Pe,Vo,Go);         {вид туризма}
  Tbez=(Bol,Neus);          {причина безделия}
  Data=record        {дата рождения}
        D:1..31;       {день}
        M:1..12;       {месяц}
        Y:1900..2100 {год}
       end;
```


GetFillPattern (Pattern) – процедура возвращает образец заполнения, установленный ранее процедурой SetFillPattern. Pattern – переменная типа FillPatternType, в которой возвращается образец заполнения. Если программа не устанавливала образец с помощью процедуры SetFillPattern, массив Pattern заполняется байтами со значением 255 ($FF).

GetFillSettings (PattInfo) – процедура возвращает текущий стиль заполнения. Здесь переменная PattInfo типа FillSettingsType, в которой возвращается текущий стиль заполнения. В модуле Graph определен тип:

```
Type
  FillSettingsType=record
                     Pattern :Word;    {образец}
                     Color   :Word     {цвет}
                   end;
```

FloodFill (x,y,*цвет_границы*) – заполняет произвольную замкнутую фигуру, используя текущий стиль заполнения до границы заданного цвета, (x,y) – произвольная точка внутри фигуры. Если фигура незамкнута, заполнение «разольется» по всему экрану.

Пример. Демонстрируем заливку маленького прямоугольника:

```
SetFillStyle(LtSlashFill,GetMaxColor);
Rectangle(0,0,8,20);
FloodFill(1,1,GetMaxColor);
```

Bar (x1,y1,x2,y2) – процедура заполняет прямоугольную область экрана текущим заполнителем. Здесь (x1,y1) – координаты левого верхнего, (x2,y2) – правого нижнего углов закрашиваемой области.

Bar3d (x1,y1,x2,y2,Depth,Top) – процедура вычерчивает трехмерное изображение параллелепипеда и закрашивает его переднюю грань. Здесь (x1,y1) – координаты левого верхнего, (x2,y2) – правого нижнего углов передней грани, Depth – переменная типа Integer, третье измерение трехмерного изображения («глубина») в пикселах, Top – переменная типа Boolean, способ изображения верхней грани. Если параметр Top имеет значение True, верхняя грань параллелепипеда вычерчивается, в противном случае – не вычерчивается. В

Образец 1								Значение байта
	■			■			■	$49
■			■			■		$92
	■			■			■	$49
■			■			■		$92
	■			■			■	$49
■			■			■		$92
	■			■			■	$49
■			■			■		$92

Образец 2								Значение байта
								$00
			■	■				$18
		■			■			$24
	■					■		$42
	■					■		$42
		■			■			$24
			■	■				$18
								$00

Закрашенный квадрат соответствует 1, незакрашенный – 0. Таким образом, получаем двоичное представление шестнадцатеричного числа, например первая строка первого образца: 01001001 – двоичное представление шестнадцатеричного числа 49.

Пример использования процедуры `SetFillPattern`. Программа рисует на экране 2 прямоугольника и заполняет их указанными образцами.

```
Uses
  Graph,Crt;
Const
  Patt1:FillPatternType=
            ($49,$92,$49,$92,$49,$92,$49,$92);
  Patt2:FillPatternType=
            ($00,$18,$24,$42,$42,$24,$18,$00);
Var
  Driver, Mode:Integer;
Begin
  Driver:=Detect;
  InitGraph(Driver,Mode,'');
  SetFillStyle(UserFill,White);
  {левый верхний квадрат}
  SetFillPattern(Patt1,1);
  Bar(0,0, GetMaxX div 2, GetMaxY div 2);
  {правый нижний квадрат}
  SetFillPattern(Patt2,2);
  Bar(GetMaxX div 2,GetMaxY div 2,GetMaxX,GetMaxY);
  ReadLn;
  CloseGraph;
End.
```



```
  Sc=record
    Fam:String[20];   {фамилия}
    Voz:Data;         {возраст}
    Class:1..12;      {класс}
    Case Zan:Tzan of
      Sport:(Vid:Tvid;Rasr:Trazr);
      Turizm:(Vidt:Tvidt;Kat:1..6);
      Bez:(Pr:Tbez)
    end;
Var
  Mas:array[1..10] of Sc;
  i,p,k,N:Byte;
{Ввод массива записей с вариантами}
Procedure Vvod;
begin
  Write ('Количество учеников N=');
  ReadLn (N);
  For i:=1 to N do
  With Mas[i] do
  begin
    Clrscr;
    Write ('Фамилия -->');
    ReadLn (Fam);
    Write ('Дата рождения -->');
    ReadLn (Voz.D,Voz.M,Voz.Y);
    Write ('Класс -->');
    ReadLn (Class);
    WriteLn ('Вид занятий:');
    WriteLn ('1 - спорт');
    WriteLn ('2 - туризм');
    WriteLn ('3 - бездельник');
    Readln (k);
    Case k of
    1: begin
         Zan:=Sport;
         Writeln ('Вид спорта:');
         WriteLn ('1 - легкая атлетика');
         WriteLn ('2 - футбол');
         WriteLn ('3 - волейбол');
         WriteLn ('4 - баскетбол');
         Readln (p);
         Case p of
```

```
      1:Vid:=L_a;
      2:Vid:=Fut;
      3:Vid:=Vol;
      4:Vid:=Bas
    end;
    Writeln ('Разряд:');
    Writeln ('1 - первый');
    Writeln ('2 - второй');
    Writeln ('3 - третий');
    Writeln ('4 - кмс');
    Writeln ('5 - мс');
    ReadLn (p);
    Case p of
      1:Rasr:=r1;
      2:Rasr:=r2;
      3:Rasr:=r3;
      4:Rasr:=kms;
      5:Rasr:=ms
    end;
  end;
2: begin
    Zan:=Turizm;
    Writeln ('Вид туризма:');
    WriteLn ('1 - пеший');
    WriteLn ('2 - водный');
    WriteLn ('3 - горный');
    Readln (p);
    Case p of
      1:Vidt:=Pe;
      2:Vidt:=Vo;
      3:Vidt:=Go
    end;
    Write ('Категория 1..6 ->');
    Readln (Kat);
  end;
3: begin
    Zan:=Bez;
    WriteLn ('Причина неучастия:');
    WriteLn ('1 - болеет');
    WriteLn ('2 - неуспевает');
    Readln (p);
    Case p of
```


`GetDefaultPalette (Palette)` – процедура возвращает структуру палитры, устанавливаемую по умолчанию. Здесь `Palette` – переменная типа `PaletteType`, в которой возвращаются размер и цвета палитры.

`SetFillStyle` (*тип_заполнения*, *цвет*) – устанавливает стиль заполнения фигур и цвет. *Тип заполнения* определяется одной из следующих констант, находящихся в модуле `Graph`:

```
Const
  EmptyFill=0;       {заполнение фоном (узор отсутствует)}
  SolidFill=1;       {сплошное заполнение}
  LineFill=2;        {заполнение - - - - }
  LtSlashFill=3;     {заполнение /////}
  SlashFill=4;       {заполнение утолщенными //////}
  BkSlashFill=5      {заполнение утолщенными \\\\\\}
  LtBkSlashFill=6;   {заполнение \\\\\\\\\}
  HatchFill=7;       {заполнение ++++++++}
  XHatchFill=8;      {заполнение xxxxxxx}
  InterleaveFill=9;  {заполнение в прямоугольную клеточку}
  WideDotFill=10;    {заполнение редкими точками}
  CloseDotFill=11;   {заполнение частыми точками}
  UserFill=12;       {узор определяется пользователем}
```

Если тип заполнения имеет значение `UserFill`, то рисунок узора определяется программистом путем обращения к процедуре `SetFillPattern`.

`SetFillPattern (Pattern,`*цвет*`)` – процедура устанавливает образец рисунка и цвет штриховки. Здесь `Pattern` – выражение типа `FillPatternType` устанавливает образец рисунка для `UserFill` в процедуре `SetFillStyle`.

Образец рисунка задается в виде матрицы из 8×8 пиксел и может быть представлен массивом из 8 байт следующего типа:

```
Type
  FillPatternType=array[1..8] of Byte;
```

Каждый разряд любого из этих байтов управляет светимостью пиксела, причем первый байт определяет 8 пиксел первой строки на экране, второй байт – 8 пиксел второй строки и т.д.

Пример двух образцов заполнения. Для каждых 8 пиксел приводится шестнадцатиричный код соответствующего байта.

GetColor – функция возвращает значение типа Word, содержащее код текущего цвета.

GetMaxColor – функция возвращает значение типа Word, содержащее максимальный доступный код цвета, который можно использовать для обращения к SetColor.

SetBkColor (*цвет*) – процедура устанавливает цвет фона.

GetBkColor – функция возвращает значение типа Word, содержащее текущий цвет фона.

SetPalette (n, Color) – процедура заменяет один из цветов палитры на новый цвет. Здесь n – номер цвета в палитре, Color – номер вновь устанавливаемого цвета.

GetPalette (PaletteInfo) – процедура возвращает размер и цвета текущей палитры. Здесь PaletteInfo – переменная типа PaletteType, возвращающая размер и цвета палитры. В модуле Graph определена константа

```
Const
  MaxColors=15;
```

и тип

```
Type
  PaletteType=record
    Size:Word; {количество цветов в палитре}
    Colors: array [0..MaxColors] of ShortInt
                    {номера входящих в палитру цветов}
                end;
```

SetAllPalette (Palette) – процедура изменяет одновременно несколько цветов палитры. Параметр Palette в заголовке процедуры описан как нетипизированный. Первый байт этого параметра должен содержать длину n палитры, остальные n байт номера вновь устанавливаемых цветов в диапазоне от –1 до MaxColors. Код –1 означает, что соответствующий цвет исходной палитры не меняется.

GetPaletteSize – функция возвращает значение типа Integer, содержащее размер палитры (максимальное количество доступных цветов).


```
              1:Pr:=Bol;
              2:Pr:=Neus
            end;
          end;
        end;
  end
end;
{Вывод массива записей с вариантами}
Procedure Vivod;
begin
  Clrscr;
  WriteLn ('Сведения о учениках:');
  WriteLn ('Фамилия     Дата рождения  Класс  Вид
занятий');
  For i:=1 to N do
  With Mas[i] do
  begin
    Write (Fam:10);
    Write (Voz.D:3,Voz.M:4,Voz.Y:5);
    Write (Class:6);
    Case Zan of
    Sport: begin
             Write (' спорт :');
             Case Vid of
               L_a:Write (' легкая атлетика ');
               Fut:Write (' футбол ');
               Vol:Write (' волейбол ');
               Bas:Write (' баскетбол ')
             end;
             Write (' Разряд:');
             Case Rasr of
               r1:Write (' первый');
               r2:Write (' второй');
               r3:Write (' третий');
               kms:Write (' кмс');
               ms:Write (' мс')
             end;
           end;
    Turizm: begin
         Write (' Туризм: ');
         Case Vidt of
           Pe: Write (' пеший');
```

```
        Vo: Write (' водный');
        Go: Write (' горный')
      end;
      Write (' Категория ',Kat);
    end;
  Bez: begin
      Write (' Бездельник:');
      Case Pr of
        Bol: Write (' болеет');
        Neus: Write (' неуспевает')
      end;
    end;
  end;
  Writeln;
 end;
end;
{Основная программа}
Begin
  Clrscr;
  Vvod;
  Vivod;
  Repeat Until KeyPressed;
End.
```


GetArcCoords (Coords) – процедура возвращает координаты трех точек: центра, начала и конца дуги. Coords – переменная типа ArcCoordsType, которая определена в модуле Graph следующим образом:

```
Type
  ArcCoordsType=record
    x, y: Integer;              {координаты центра}
    Xstart, Ystart: Integer; {начало дуги}
    Xend, Yend: Integer       {конец дуги}
  end;
```

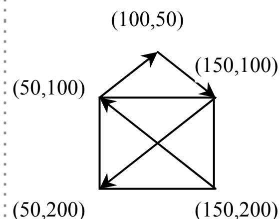
Ellipse (x,y,*начальный_угол*, *конечный_угол*, x_*радиус*, y_*радиус*) – процедура вычерчивает эллипсную дугу с центром в точке (x,y) от начального до конечного угла заданными горизонтальным *x-радиусом* и вертикальным *y-радиусом*.

Краски, палитра, заполнение

SetColor (*цвет*) – процедура устанавливает цвет выводимых линий и символов. Цвет определяется одной из следующих констант, находящихся в модуле Graph:

```
Const
  Black=0;           {черный}
  Blue=1;            {синий}
  Green=2;           {зеленый}
  Cyan=3;            {голубой}
  Red=4;             {красный}
  Magenta=5;         {сиреневый}
  Brown=6;           {коричневый}
  LightGray=7;       {светло-серый}
  DarkGray=8;        {темно-серый}
  LightBlue=9;       {голубой}
  LightGreen=10;     {светло-зеленый}
  LightCyan=11;      {светло-голубой}
  LightRed=12;       {светло-красный}
  LightMagenta=13; {светло-сиреневый}
  Yellow=14;         {желтый}
  White=15;     {белый}
```

```
Uses Graph,Crt;
Type
 Tmas=array [1..10]
        of PointType;
{определяем массив записей, содержащий координаты вершин фигуры}
Const
 mas:Tmas=((x:150;y:200),
           (x:50;y:100),
           (x:100;y:50),
           (x:150;y:100),
           (x:50;y:200),
           (x:50;y:100),
           (x:150;y:100),
           (x:150;y:200),
           (x:50;y:200),
           (x:150;y:200));
Var
  Driver, Mode:Integer;
Begin
  Driver:=Detect;
  InitGraph(Driver,Mode,'');
  SetLineStyle(0,0,3);
  {рисуем фигуру}
  DrawPoly(10,mas);
  ReadLn;
  CloseGraph;
End.
```



Стрелками указан начальный порядок обхода вершин

Дуги, окружности, эллипсы

`Circle (x,y,`*радиус*`)` – процедура вычерчивает окружность с центром в точке (x,y) заданного радиуса.

`Arc (x,y,`*начальный_угол*`,`*конечный_угол*`,`*радиус*`)` – процедура чертит дугу окружности с центром в точке (x,y) от начального до конечного угла заданного радиуса. Углы отсчитываются против часовой стрелки и указываются в градусах. Нулевой угол соответствует горизонтальному направлению вектора слева направо.

ФАЙЛЫ

Под *файлом* понимается либо поименованная область внешней памяти персонального компьютера (жесткого диска, гибкой дискеты), либо логическое устройство – потенциальный источник или приемник информации.

Любой файл имеет три характерные особенности:

1. У файла есть имя, что дает возможность программе работать одновременно с несколькими файлами.
2. Файл содержит компоненты одного типа. Типом компонент может быть любой тип, кроме файлов.
3. Длина вновь создаваемого файла никак не оговаривается при его объявлении и ограничивается только емкостью устройств внешней памяти.

Файловый тип или переменную файлового типа можно задать одним из трех способов:

имя_файлового_типа=**`File of`** *тип_компонент;*

имя_файлового_типа =**`Text`**;

имя_файлового_типа =**`File`**;

которым соответствуют три вида файлов:

- типизированный файл,
- текстовый файл,
- нетипизированный файл.

Доступ к файлам

Любой программе доступны два предварительно объявленных файла со стандартными файловыми переменными: `INPUT` – для чтения данных с клавиатуры и `OUTPUT` – для вывода на экран.

Логические устройства

Стандартные аппаратные средства персонального компьютера, такие как клавиатура, экран дисплея, принтер и коммуникационные каналы ввода-вывода, определяются специальными именами, которые называются *логическими устройствами.* Все они рассматриваются как потенциальные источники или приемники текстовой информации.

`CON` – логическое имя, которое определяет клавиатуру или экран дисплея. Ввод с клавиатуры буферизируется: символы по мере нажатия на клавиши помещаются в специальный строковый буфер, который пе-

редается программе только после нажатия клавиши *Enter*. Буферизация ввода дает возможность редактирования вводимой строки.

PRN – логические имя принтера. Если подключено несколько принтеров, доступ к ним осуществляется по логическим именам LPT1, LPT2, LPT3. Стандартный библиотечный модуль PRINTER объявляет имя файловой переменной LST и связывает его с логическим устройством LPT1.

AUX – логическое имя коммуникационного канала, который обычно используется для связи компьютера с другими машинами. Как правило, в составе персонального компьютера имеются два коммуникационных канала, которым даются имена логических устройств COM1 и COM2.

NUL – логическое имя «пустого» устройства. Это устройство чаще всего используется в отладочном режиме и трактуется как устройство-приемник информации неограниченной емкости.

Процедуры для работы с любыми файлами

Assign (*файловая_переменная,имя_файла*) – связать файловую переменную с именем файла.

Close (*файловая_переменная*) – закрыть файл.

Rename (*файловая_переменная, новое_имя*) – переименовать файл.

Erase (*файловая_переменная*) – уничтожить файл.

Flush (*файловая_переменная*) – очищает внутренний буфер файла и, таким образом гарантирует сохранность всех последних изменений файла на диске. Процедура игнорируется, если файл был открыт для чтения процедурой Reset.

ChDir (*путь*) – изменение текущего каталога. Путь – это строковое выражение, содержащее путь к устанавливаемому по умолчанию каталогу.

GetDir (*устройство*, *каталог*) – определяет имя текущего каталога. Устройство – выражение типа Word, содержащее номер устройства: 1 – диск A, 2 – диск B и т.д. Каталог – переменная строкового ти-

способ взаимодействия выводимых линий с изображением. Если параметр *режим* имеет значение 0, выводимые линии накладываются на существующее изображение обычным образом. Если значение 1, то это наложение осуществляется с применением логической операции Xor (исключающее или): в точках пересечения выводимой линии с имеющимся на экране изображением светимость пиксел инвертируется на обратную, так что два следующих друг за другом вывода одной и той же линии на экран не изменяет его вид. Режим, установленный процедурой SetWriteMode, распространяется на процедуры Drawpoly, Line, LineTo, Rectangle. Для задания параметра *режим* можно использовать следующие определенные в модуле константы:

```
Const
  CopyPut=0;
  XorPut=1;
```

Пример изменения стиля линии:

```
SetLineStyle(Solid,0,ThickWidth);
Line(1,1,640,350);
```

Многоугольники

Rectangle (x1,y1,x2,y2) – процедура вычерчивает прямоугольник с заданными координатами углов. Здесь (x1,y1) – координаты левого верхнего, (x2,y2) – правого нижнего углов прямоугольника. Прямоугольник вычерчивается с использованием текущего цвета и стиля линий.

DrawPoly (n,Points) – процедура вычерчивает произвольную ломаную линию, заданную координатами точек излома. Здесь n – количество точек излома, включая обе крайние точки, Points – переменная типа PointType, содержащая координаты точек излома. Координаты точек излома задаются парой значений типа Word: первое определяет горизонтальную, второе – вертикальную координаты. Для них можно использовать следующий определенный в модуле тип:

```
Type
  PoitType=record
    x,y:Word
  end;
```

При вычерчивании используется текущий цвет и стиль линий.

Пример. С помощью ломаной линии нарисовать заданную фигуру.

SetLineStyle (*тип, образец, толщина_линии*) – процедура устанавливает стиль вычерчиваемых линий.

Тип линии может быть задан с помощью одной из следующих констант:

```
Const
  SolidLn=0;    {Сплошная линия}
  DottedLn=1;   {Точечная линия}
  CenterLn=2;   {Штрих-пунктирная линия}
  DashedLn=3;   {Пунктирная линия}
  UserBitLn=4;  {Узор линии определяется пользователем}
```

Образец учитывается только для линий, вид которых определяется пользователем. При этом два байта параметра *образец* определяют образец линии: каждый установленный в единицу бит этого слова соответствует светящемуся пикселу в линии, нулевой бит – несветящемуся пикселу. Таким образом, параметр *образец* задает отрезок линии длиной в 16 пиксел. Этот образец периодически повторяется по всей длине линии.

Параметр *толщина линии* может принимать одно из двух значений:

```
Const
  NormWidth=1;      {Толщина в один пиксел}
  ThickWidth=3;     {Толщина в три пиксела}
```

Установленный процедурой стиль линии используется при построении прямоугольников, многоугольников и других фигур.

GetLineSettings (StyleInfo) – процедура возвращает текущий стиль линий. Параметр StyleInfo – переменная типа LineSettingsType, в которой возвращается текущий стиль линий. Тип LineSettingsType определен в модуле Graph следующим образом:

```
Type
  LineSettingsType=record
    LineStyle:Word;    {тип линии}
    Pattern:Word;      {образец}
    Tickness:Word;     {толщина}
  end;
```

SetWriteMode (*режим*) – процедура устанавливает способ взаимодействия вновь выводимых линий с уже существующим на экране изображением. *Режим* – выражение типа Integer, задающее

па, в которой возвращается путь к текущему каталогу на указанном диске.

MkDir (*каталог*) – создает новый каталог на указанном диске. каталог – переменная строкового типа, задающая путь к каталогу.

RmDir (*каталог*) – удаляет каталог. Удаляемый каталог должен быть пустым.

FindFirst (*маска*, *атрибуты*, *имя*) – возвращает атрибуты первого из файлов, зарегистрированных в указанном каталоге.

Маска – строковое выражение, содержащее маску файл, например *a?.pas*, **.dat*. Маске может предшествовать путь.

Атрибуты – выражение типа Byte, содержащее уточнение к маске. В модуле Dos.tpu определены следующие файловые атрибуты:

ReadOnly – только чтение,
Hidden – скрытый файл,
SysFile – системный файл,
VolumeID – идентификатор тома,
Directory – имя подкаталога,
Archive – архивный файл,
AnyFile – любой файл.

Имя – переменная типа SearchRec, в которой будет возвращено имя файла. Этот тип в модуле Dos.tpu определяется следующим образом:

```
Type
  SearchRec=record
    Fill:array[1..21] of Byte;
    Attr:Byte;       {атрибуты файла}
    Time:LongInt;    {время создания или последнего обновления}
                     {файла; возвращается в упакованном формате}
    Size:LongInt;    {длина файла в байтах}
    Name:String[12]  {имя и расширение файла}
        end;
```

Для распаковки параметра Time используется процедура:

```
UnPackTime (Time:LongInt; var T:DateTime);
```

В модуле Dos.tpu объявлен следующий тип DateTime:

```
Type
  DateTime=record
    year :Word; {год в формате XXXX}
```

```
    month:Word; {месяц 1..12}
    day  :Word; {день 1..31}
    hour :Word; {час 0..23}
    min  :Word; {минуты 0..59}
    sec  :Word; {секунды 0..59}
  end;
```

Результат обращения к процедуре FindFirst можно проконтролировать с помощью функции DosError типа Word, которая возвращает значения:

0 – нет ошибок;

2 – не найден каталог;

18 – каталог пуст (нет указанных файлов).

FindNext (*следующий_файл*) – возвращает имя следующего файла в каталоге.

Пример. Вывести на экран список всех pas-файлов текущего каталога.

```
Uses Dos;
Var
  S:SearchRec;
Begin
  FindFirst('*.pas',AnyFile,S);
  While DosError=0 do
  begin
    With S do
      WriteLn(Name:12;Size:12);
    FindNext(S)
  end
End.
```

GetfTime (*файловая_переменная*, *время*) – возвращает время создания или последнего обновления файла. Время – переменная типа LongInt. Время возвращается в упакованном формате.

SetfTime (*файловая_переменная*, *время*) – устанавливает новую дату создания или обновления файла. Время – переменная типа LongInt, указывающая дату и время в упакованном формате.

Упаковать запись типа DateTime в переменную типа LongInt можно процедурой

```
PackTime (var T:DateTime; var Time:LongInt);
```



```
  ReadLn;
  SetVisualPage(1);
  ReadLn;
  SetVisualPage(0);
  ReadLn;
  CloseGraph;
End.
```

С помощью оператора

```
If Driver<>HercMono then SetGraphMode(Mode-1);
```

устанавливается многостраничный режим работы на адаптерах EGA, MCGA, VGA. После инициализации графики с Driver=Detect устанавливается режим работы с максимально возможным номером; перечисленные адаптеры могут работать только с одной графической страницей, чтобы обеспечить работу с двумя страницами, следует уменьшить номер режима.

Линии и точки

PutPixel (x,y,*цвет*) – процедура выводит заданным цветом точку с координатами (x,y). Координаты задаются относительно левого верхнего угла окна или, если окно не установлено, относительно левого верхнего угла экрана.

GetPixel (x,y) –функция возвращает значение типа Word, содержащее цвет пиксела с указанными координатами.

Line (x1,y1,x2,y2) – процедура вычерчивает линию с координатами начала (x1,y1) и конца (x2,y2). Линия вычерчивается текущим стилем и текущим цветом.

LineTo (x,y) – процедура вычерчивает линию от текущего положения указателя до точки с заданными координатами (x,y). Линия вычерчивается текущим стилем и текущим цветом.

LineRel (dx,dy) – процедура вычерчивает линию от текущего положения указателя до положения, заданного приращениями координат (dx,dy). Линия вычерчивается текущим стилем и текущим цветом.

GetMaxX – функция возвращает максимальную координату экрана по горизонтали в текущем режиме работы.

GetMaxY – функция возвращает максимальную координату экрана по вертикали в текущем режиме работы.

GetX – функция возвращает текущую координату указатели по горизонтали.

GetY – функция возвращает текущую координату указатели по вертикали.

SetActivePage (*номер_страницы*) – процедура делает активной указанную страницу видеопамяти. Фактически процедура просто переадресует графический вывод в другую область видеопамяти. Активная страница может быть невидимой. Нумерация страниц начинается с нуля.

SetVisualPage (*номер_страницы*) – процедура делает видимой страницу с указанным номером. Нумерация страниц начинается с нуля.

Пример работы с видеостраницами. Программа сначала рисует квадрат в видимой странице и окружность – в невидимой. После нажатия *Enter* происходит смена видимых страниц.

```
Uses Graph,Crt;
Var
  Driver, Mode:Integer;
Begin
  Driver:=Detect;
  InitGraph(Driver,Mode,'');
  If Driver<>HercMono then
    SetGraphMode(Mode-1);
  SetActivePage(0);
  {заполняем видимую страницу}
  Rectangle(10,10,GetMaxX div 2, GetMaxY div 2);
  OutTextXY(0,0,'Page 0. Press Enter...');
  {заполняем невидимую страницу}
  SetActivePage(1);
  Circle(GetMaxX div 2, GetMaxY div 2, 100);
  OutTextXY(0,0,'Page 1. Press Enter...');
  {переключаемся между страницами}
```


GetfAttr(*файловая_переменная*, *атрибуты*) – позволяет получить атрибуты файла.

SetfAttr(*файловая_переменная*, *атрибуты*) – позволяет установить атрибуты файла.

Fsplit (*файл, путь, имя, расширение*) – «расщепляет» имя файла, т.е. возвращает в качестве отдельных параметров путь к файлу, его имя и расширение. Процедура не проверяет наличие на диске указанного файла.

Функция для работы с любыми файлами

EOF (*файловая_переменная*) – функция возвращает значение TRUE, если указатель файла стоит в конце файла.

IOResult возвращает условный признак последней операции ввода-вывода. Если операция завершилась успешно, функция возвращает значение ноль. Она становится доступной только при отключенном автоконтроле ошибок ввода-вывода. Директива компилятора {$I-} отключает, а директива {$I+} включает автоконтроль.

```
  Assign (F,Name);
  {$I-}
  Reset(F);
  If IOResult<>0 then Halt;
  Close(F);
  {$I+}
```

DiskFree (*диск*) – функция возвращает значение типа LongInt объема в байтах свободного пространства на указанном диске. Диск – выражение типа Byte, определяющее номер диска: 0 – устройство по умолчанию, 1 – диск A, 2 – диск B и т.д. Функция возвращает значение –1, если указан номер несуществующего диска.

DiskSize (*диск*) – функция возвращает значение типа LongInt полного объема в байтах указанного диске или –1, если указан номер несуществующего диска.

Fsearch (*имя_файла*, *список_каталогов*) – ищет файл в списке каталогов. Имя и список каталогов – строковые выражение. Результат поиска возвращается в виде строки типа PathStr.

```
Type
  PathStr=String[79];
```

Fexpand (*файл*) – функция дополняет файловое имя до полной спецификации, т.е. с указанием устройства и пути. Файл – строковое выражение или переменная типа PathStr.

Типизированные файлы

Формат описания:
1-ый способ:

```
Type
  имя_файлового_типа=File of тип_компонент;
Var
  файловая_переменная: имя_файлового_типа;
```

2-ой способ:

```
Var
  файловая_переменная : File of тип_компонент;
```

Пример. Если компонентами файла являются записи, то он описывается следующим образом:

```
Type
  Sved=record
         Fio:String;
         Nom:String[10];
         b1,b2,b3:Byte;
       end;
Var
  Fv:File of Sved;      {переменная доступа к файлу}
  Rv:Sved;              {переменная доступа к записи}
```

Если компонентами файла являются целые числа, то он описывается так:

```
Var
  F:File of Integer;
```

Доступ к компонентам файла осуществляется через указатель файла (файловую переменную). В ней хранится текущий номер компоненты файла.

Существует 2 способа доступа к компонентам файла:

1. последовательный


```
Type
  ViewPortType=record
    x1,y1,x2,y2 :Integer;     {координаты окна}
    Clip        :Boolean;     {признак отсечки}
  end;
```

MoveTo (x,y) – процедура устанавливает новое положение указателя в позиции (x,y). Координаты определяются относительно левого верхнего угла окна или, если окно не установлено, экрана.

MoveRel (dx,dy) – процедура устанавливает новое положение указателя в относительных координатах (dx,dy). Здесь dx, dy – приращения новых координат указателя соответственно по горизонтали и вертикали. Приращения задаются относительно того положения, которое занимал указатель к моменту обращения к процедуре.

ClearDevice – процедура очищает графический экран. После обращения к процедуре указатель устанавливается в левый верхний угол экрана, а сам экран заполняется цветом фона, заданным процедурой SetBkColor.

ClearViewPort – процедура очищает графическое окно, а если окно не определено к этому моменту – весь экран. Указатель перемещается в левый верхний угол окна.

GetAspectRatio (x,y) – процедура возвращает значения x,y, позволяющие оценить соотношение сторон графического экрана в пикселах. Найденный с их помощью коэффициент может использоваться для построения правильных геометрических фигур.

Пример. Для построения квадрата со стороной n пиксел по вертикали, нужно использовать операторы:

```
GetAspectRatio (Xasp,Yasp);
Rectangle(x1,y1,x1+n*round(Yasp/Xasp),y1+n);
```

Если же n определяет длину квадрата по горизонтали, используется оператор

```
Rectangle(x1,y1,x1+n,y1+n*round(Xasp/Yasp));
```

SetAspectRatio (x,y) – процедура устанавливает масштабный коэффициент отношения сторон графического экрана. Здесь x,y – устанавливаемые соотношения сторон.

DetectGraph (*драйвер*, *режим_работы*) – процедура возвращает тип драйвера и максимально возможный режим его работы.

GetDriverName – функция возвращает значение типа String, содержащее имя загруженного графического драйвера.

GetMaxMode – функция возвращает значение типа Integer, содержащее количество возможных режимов работы адаптера.

GetModeName (*номер_режима*) – функция возвращает значение типа String, содержащее разрешение экрана и имя режима работы адаптера по его номеру.

GetModeRange (*тип_адаптера*, *min*, *max*) – процедура возвращает диапазон возможных режимов работы заданного графического адаптера. *Тип адаптера* – число типа Integer, *min*, *max* – переменные типа Integer, в которых возвращаются нижнее и верхнее возможные значения номера режима.

Координаты окна, страницы

SetViewPort (x1,y1,x2,y2,ClipOn) – процедура устанавливает прямоугольное окно на графическом экране, (x1,y1) – координаты левого верхнего, (x2,y2) – координаты правого нижнего угла окна. Координаты окна всегда задаются относительно левого верхнего угла экрана. ClipOn – логическое выражение, определяющее «отсечку» не умещающихся в окне элементов изображения. Если параметр ClipOn=True, элементы изображения, не умещающиеся в переделах окна, отсекаются, в противном случае границы окна игнорируются. Для управления этим параметром можно использовать следующие определенные в модуле константы:

```
Const
  ClipOn=True;    {Включить отсечку}
  ClipOff=False;  {Не включать отсечку}
```

GetViewSettings (ViewInfo) – процедура возвращает координаты и признак отсечки текущего графического окна. Параметр ViewInfo – переменная типа ViewPortType, который определен в модуле Graph следующим образом:

2. произвольный

При последовательном доступе поиск компоненты начинается с начала файла и проверяется по очереди до нужной компоненты.

Произвольный доступ позволяет обращаться к компонентам файла по их порядковому номеру. Для организации произвольного доступа используется процедура Seek.

Процедуры для работы с типизированными файлами

Rewrite (*файловая_переменная*) – открыть новый файл для записи.

Reset (*файловая_переменная*) – открыть существующий файл для чтения. Разрешается обращаться к *типизированным* файлам, открытым процедурой Reset, с помощью процедуры Write для записи информации в файл.

Read (*файловая_переменная*, *переменная_доступа_к_компонентам*) – считывает компонент из файла.

Write (*файловая_переменная*, *переменная_доступа_к_компонентам*) – записывает компонент в файл.

Seek (*файловая_переменная*, *номер_компоненты*) – смещает указатель файла к требуемому компоненту. Первый компонент файла имеет номер ноль.

Функции для работы с типизированными файлами

FileSize (*файловая_переменная*) – возвращает значение типа Longint, которое содержит количество компонент файла.

FilePos (*файловая_переменная*) возвращает значение типа Longint, которое содержит порядковый номер компонента файла, который будет обрабатываться следующей операцией ввода-вывода.

Порядок создания файла

1. Связать файловую переменную с именем файла (Assign).
2. Открыть новый файл (Rewrite)
3. Записать компонент в файл (Write)
4. Закрыть файл (Close).

Порядок использования файла

1. Связать файловую переменную с именем файла (`Assign`).
2. Открыть существующий файл (`Reset`)
3. Прочитать компонент из файла (`Read`)
4. Закрыть файл (`Close`).

Если компонентами файла являются записи, то применяются следующие виды корректировок файла:
1. Расширение файла за счет внесения новых компонент.
2. Полная замена содержимого записи.
3. Корректировка значений полей отдельных записей.
4. Удаление компонент из файла.

Порядок расширения файла за счет внесения новых компонент

1. Связать файловую переменную с именем файла (`Assign`).
2. Открыть существующий файл (`Reset`)
3. Установить указатель файла за последним компонентом файла `Seek(`*файловая_переменная*`,FileSize(`*файловая_переменная*`))`
4. Записать компонент в файл (`Write`)
5. Закрыть файл (`Close`).

Порядок замены содержимого записи

1. Связать файловую переменную с именем файла (`Assign`).
2. Открыть существующий файл (`Reset`)
3. Установить указатель файла перед компонентом с нужным номером `Seek(`*файловая_переменная*`,`*номер_компоненты*`)`
4. Прочитать компоненту из файла (`Read`)
5. Установить указатель файла перед компонентом с нужным номером `Seek(`*файловая_переменная*`,`*номер_компоненты*`)`
6. Записать компонент в файл (`Write`)
7. Закрыть файл (`Close`).

Пример. Создать файл записей, содержащих сведения о сдаче студентами сессии. Структура записи: фамилии студента, номер группы, результаты сдачи трех экзаменов. Распечатать список студентов, получающих стипендию. Условие получения стипендии – средний балл больше 5. Предусмотреть все виды корректировки файла.

`GraphErrorMsg (`*код_ошибки*`)` – функция возвращает значение типа `String`, в котором по указанному коду ошибки дается соответствующее текстовое сообщене. *Код ошибки* – это значение, возвращаемое функцией `GraphResult`.

`RestoreCRTMode` – процедура служит для кратковременного возврата в текстовый режим.

`GetGraphMode` – функция возвращает значение типа Integer, в котором содержится код установленного режима работы графического адаптера.

`SetGraphMode (`*режим*`)` – процедура устанавливает новый графический режим работы адаптера. *Режим* – число типа `Integer`, задающее режим работы адаптера.

Пример перехода из графического режима в текстовый и обратно.

```
Uses Graph;
Var
  Driver, Mode, Error: Integer;
Begin
  {инициируем графический режим}
  Driver:=Detect;
  InitGraph(Driver,Mode,'');
  Error:=GraphResult;      {запоминаем результат}
  If Error<>grOK then      {проверяем ошибку}
    WriteLn(GraphErrorMsg(Error))  {есть ошибка}
  else              {нет ошибки}
    Begin
      {это графический режим}
      Repeat Until KeyPressed;
      {переходим в текстовый режим}
      RestoreCRTMode;
      Repeat Until KeyPressed;
      {возвращаемся в графический режим}
      SetGraphMode(GetGraphMode);
      Repeat Until KeyPressed;
      CloseGraph; {закрыли графический режим}
    end;
End.
```

ГРАФИЧЕСКИЕ ВОЗМОЖНОСТИ ЯЗЫКА Pascal

Процедуры и функции работы с графикой

Инициализация графического режима

`InitGraph` (*драйвер*, *режим*, *путь*) – процедура инициирует графический режим. *Драйвер* – переменная типа `Integer`, определяет тип графического драйвера. *Режим* – переменная типа `Integer`, задающая режим работы графического адаптера. *Путь* – выражение типа `String`, содержащее имя файла драйвера и, возможно, маршрут его поиска. Процедура загружает графический драйвер в оперативную память и переводит адаптер в графический режим работы. Тип драйвера должен соответствовать типу графического адаптера. Обычно при инициализации графики в качестве драйвера указывается значение `Detect` – режим автоопределения типа графического драйвера. В этом случае режим работы графического адаптера определяется по умолчанию.

`Closegraph` – процедура завершает работу адаптера в графическом режиме и восстанавливает текстовый режим работы экрана.

Пример. Порядок перехода в графический режим.

```
Uses Graph;
Var
  Driver, Mode: Integer;
Begin
    Driver:=Detect;
    {режим автоопределения типа графического драйвера}
    InitGraph(Driver,Mode,'');
    {графический режим}
    Repeat Until KeyPressed; {задержка экрана}
    CloseGraph; {закрыли графический режим}
End.
```

`GraphResult` – функция возвращает значение типа `Integer`, в котором закодирован результат последнего обращения к графическим процедурам. Если ошибка не обнаружена, значением функции будет ноль, в противном случае – отрицательное число, соответствующее значению одной из зарезервированных констант ошибок. Например,

```
    Const grOk=0; {Нет ошибок}
```



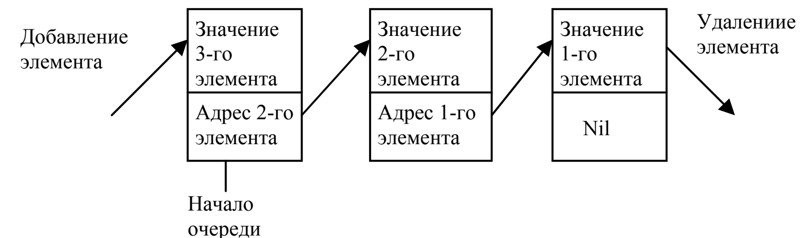
```
Program Zapisi;
  Uses Crt;
  Const
    FileName:String[10]='Stud.dat'; {имя Файла}
  Type Sved=record
         Fio:String[50]; {Фамилия}
         Nom:String[10]; {Номер группы}
         b1,b2,b3:0..10; {Результаты сдачи экзаменов}
         sb:Real;        {Средний балл}
       end;
  Var
    Fv,Fv1:File of Sved;
    Rv: Sved;
    i,N:Byte;
{Создание файла записей}
Procedure Vvod;
Begin
  Assign (Fv,FileName);
  Rewrite(Fv);
  While True do
  With Rv do
  begin
    Clrscr;
    Write ('ФИО (признак окончания ввод ***)-->');
    ReadLn(Fio);
    If Fio='***' then
    begin
      Close(Fv);
      Exit;
    end;
    Write ('Группа -->');
    ReadLn(Nom);
    Write ('Оценки -->');
    ReadLn(b1,b2,b3);
    sb:=(b1+b2+b3)/3;
    Write(Fv,Rv);
  end
end;
{Вывод исходного файла записей}
Procedure Vivod;
begin
  clrscr;
```

```
  WriteLn ('Сведения о студентах:');
  WriteLn ('Фамилия  Группа Оценки  Средний балл');
  Assign (Fv,FileName);
  Reset(Fv);
  While not EOF(Fv) do
  With Rv do
  begin
    Read(Fv,Rv);
    WriteLn (Fio:10,Nom:10,b1:2,b2:2,b3:2,sb:5:1);
  end;
  Close (Fv);
  Repeat Until KeyPressed;
end;
{Вывод списка студентов, получающих стипендию}
Procedure Obr;
begin
  clrscr;
  WriteLn ('Сведения о студентах:');
  WriteLn ('Фамилия  Группа Оценки  Средний балл');
  Assign (Fv,FileName);
  Reset(Fv);
  While not EOF(Fv) do
  With Rv do
  begin
    Read(Fv,Rv);
    If sb>5 then
      WriteLn(Fio:10,Nom:10,b1:2,b2:2,b3:2,sb:5:1);
  end;
  Close (Fv);
  Repeat Until KeyPressed;
end;
{Расширение файла за счет внесения новых компонент}
Procedure Rasch;
Begin
  Assign (Fv,FileName);
  Reset(Fv);
  Seek(Fv,FileSize(Fv));
  While True do
  With Rv do
  begin
    Clrscr;
    Write ('ФИО (признак окончания ввод ***)-->');
```



```
Begin
  Kon:=Top;
  While Kon<>Nil do {пока не дошли до конца}
  begin
    WriteLn (Kon^.Inf); {выводим значение элемента}
    Kon:=Kon^.Next; {смещаем указатель на следующий
                     элемент}
  end
End;
{Основная программа}
Begin
  Sozd;
  Dobav;
  Udal;
  Rasp;
End.
```

2. Очередь – это структура данных, в один конец которой добавляются элементы, а с другого конца извлекаются. Принцип работы очереди FIFO (First In First Out) – «первым пришел, первым вышел». Для организации такой структуры используются две переменные: для указания начала и конца очереди, например `Left` и `Right`. При добавлении элемента в очередь он располагается в памяти в соответствии со значением `Right`, а значение `Right` изменяется и указывает на следующее свободное место памяти. Выборка элемента из очереди происходит исходя из значения n, которое изменяется и указывает на следующий элемент очереди. Когда очередь пуста, значения `Left` и `Right` равны. Как и для стека, для очереди определены операции занесения элемента в очередь и его извлечения из очереди. Схема работы очереди:



```
{тип элементов стека – Integer, ввод элементов прекращается, если
введено значение 999}
Begin
  Top:=Nil;
  While True do
  begin
    ReadLn(Value);
    If Value=999 then Exit;
    New(Kon);          {Выделили память}
    Kon^.Next:=Top;    {В дополнительную часть занесли адрес
       предыдущего элемента, для первого элемента значение Nil}
    Kon^.Inf:=Value {В дополнительную часть занесли
                      значение введенного элемента}
    Top:=Kon; {Запомнили адрес введенного элемента}
  end
End;
{Добавление элементов в стек}
Procedure Dobavl;
{тип элементов стека – Integer, ввод элементов прекращается, если
введено значение 999}
Begin
  While True do
  begin
    ReadLn(Value);
    If Value=999 then Exit;
    New(Kon);          {Выделили память}
    Kon^.Next:=Top;    {В дополнительную часть занесли адрес
                        предыдущего элемента}
    Kon^.Inf:=Value {В дополнительную часть занесли
                      значение введенного элемента}
    Top:=Kon; {Запомнили адрес введенного элемента}
  end
End;
{Удаление элемента стека}
Procedure Udal;
Begin
  Top:=Top^.Next; {смещаем указатель на предыдущий элемент}
End;
{Просмотр элементов стека}
Procedure Pasp;
{Вывод элементов стека происходит в обратном порядке}
```



```
    ReadLn(Fio);
    If Fio='***' then
    begin
      Close(Fv);
      Exit;
    end;
    Write ('Группа -->');
    ReadLn(Nom);
    Write ('Оценки -->');
    ReadLn(b1,b2,b3);
    sb:=(b1+b2+b3)/3;
    Write(Fv,Rv);
  end
end;
{Замена содержимого записи}
Procedure Zam_zapisi;
Begin
  Clrscr;
  Assign (Fv,FileName);
  Reset(Fv);
  Write ('Введите номер заменяемой записи N=');
  ReadLn (N);
  Seek(Fv,N-1);
  Read(Fv,Rv);
  Write ('ФИО: ',Fio,'-->');
  ReadLn(Fio);
  Write ('Группа: ',Nom,' -->');
  ReadLn(Nom);
  Write ('Оценки: ',b1:2,b2:2,b3:2,' -->');
  ReadLn(b1,b2,b3);
  sb:=(b1+b2+b3)/3;
  Seek(Fv,N-1);
  Write(Fv,Rv);
  Close(Fv);
end;
{Удаление записи из файла}
Procedure Udal_zapisi;
Begin
  Clrscr;
  Assign (Fv,FileName);
  Reset(Fv);
  Write ('Введите номер удаляемой записи N=');
```

```
  ReadLn (N);
  Assign (Fv1,'Temp.dat'); {создаем промежуточный файл}
  Rewrite(Fv1);              {открываем его для записи}

  For i:=1 to N-1 do {считываем из исходного файла и записываем в промежуточный файл все компоненты до удаляемой}
  begin
    Read(Fv,Rv);
    Write(Fv1,Rv);
  end;
  Read(Fv,Rv); {считываем удаляемую компоненту}
               { но не записываем}
  While not EOF(Fv) do {считываем все остальные}
                       {компоненты до конца файла}
  begin
    Read(Fv,Rv);
    Write(Fv1,Rv);
  end;
  Close(Fv);
  Close(Fv1);
  {переписываем данные из промежуточного файла в исходный}
  Assign (Fv,FileName);
  Rewrite(Fv);
  Assign (Fv1,'Temp.dat');
  Reset(Fv1);
  While not EOF(Fv1) do
  begin
    Read(Fv1,Rv);
    Write(Fv,Rv);
  end;
  Close(Fv);
  Close(Fv1);
end;
{Основная программа}
Begin
  Vvod;
  Vivod;
  Obr;
  Rasch;
  Zam_zapisi;
  Udal_zapisi;
End.
```


вается в переменной, которая всегда присутствует в программе обработки списков. Если в списке нет элементов, т.е. список пустой, последний элемент содержит значение Nil.

Основными операциями, которые можно выполнять над списками, являются операции включения записи в список и удаления её из списка.

Наибольшее распространение получили следующие виды списков: стеки, очереди.

1. Стек – это список с одной точкой доступа к его элементам, которая называется вершиной стека. Добавить или убрать элемент можно только через его вершину. Принцип работы стека – LIFO (Last In First Out), т.е. последний вошел, первый вышел. Для стека определены операции занесения элемента в стек и извлечении элемента из стека. Операция занесения элемента в стек определяется только значением элемента. Извлечение элемента заключается в присвоении переменной значения первого элемента стека и удалении этого элемента.

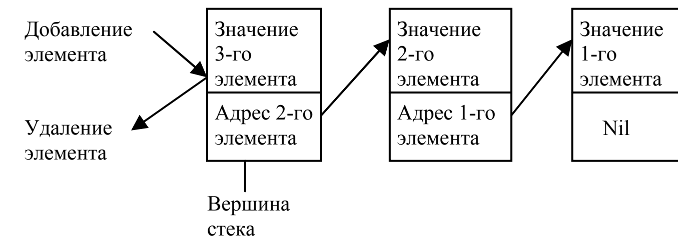


Схема работы стека

Пример программы работы со стеком: формирование стека, добавление, удаление и просмотр элементов.

```
Program Demo_Stack;
Type
  Ukaz=^Stack;
  Stack=record
    Inf:Integer;   {информационная часть}
    Next:Ukaz      {дополнительная часть}
  end;
Var
  Top,Kon:Ukaz;
  Value:Integer;
{организация стека}
Procedure Sozd;
```

Использование указателей для обработки массивов

Продемонстрируем работу указателей на примере обработки одномерного массива: умножим все элементы массива на число.

```
Program Demo_ukaz;
Type
  mas=array[1..7000] of Integer;
Var
  A:^mas;        {указатель для работы с массивом}
  c:^Integer;  {указатель для работы с числом}
  i,n:Integer;
Begin
  Write('Размерность массива n=');
  ReadLn (n);
  New(A);  {выделили память для массива}
  New(c);  {выделили память для числа}
  Write('Введите число c=');
  ReadLn (c^);
  WriteLn('Введите элементы вектора A:');
  For i:=1 to n do
    ReadLn(A^[i]);
  WriteLn ('Вектор A:');
  For i:=1 to n do
    Write(A^[i]:3);
  WriteLn;
  WriteLn ('Вектор cA:');
  For i:=1 to n do
    Write(A^[i]*c^:3);
  Dispose(A); {освобождаем память}
  Dispose(c);
End.
```

Использование указателей для работы со списками

Указатели являются эффективным средством построения списков.

Списком называется упорядоченная структура, каждый элемент которой содержит ссылку, связывающую его со следующим элементом. Для организации списков используются записи, состоящие из двух смысловых частей: основной и дополнительной. Основная часть содержит подлежащую обработке информацию, в дополнительной находится указатель на следующую запись списка. Начало списка указы-

Текстовые файлы

Текстовый файл – это файл, состоящий из компонент, являющихся строками. Текстовый файл трактуется Turbo Pascal как совокупность строк переменной длины. Доступ к каждой строке возможен лишь последовательно, начиная с первой. При создании текстового файла в конце каждой строки ставится специальный признак `EOLN` (End Of Line – конец строки), а в конце всего файла – признак `EOF` (End Of File – конец файла).

Формат описания:

1-ый способ:

Type
  имя_файлового_типа=**Text**;
Var
  файловая_переменная: имя_файлового_типа;

2-ой способ:

Var
  файловая_переменная : **Text**;

Процедуры для работы с текстовыми файлами

`Rewrite` (*файловая_переменная*) – открыть новый файл для записи.

`Reset` (*файловая_переменная*) – открыть существующий файл для чтения.

`Append` (*файловая_переменная*) – открыть существующий файл для дополнения. Процедура открывает файл для записи и устанавливает указатель файла в конец файла.

`Read` (*файловая_переменная*, *список_ввода*) – обеспечивает ввод символов, строк и чисел. Список ввода – последовательность из одной или более переменных типа `Char`, `String`, а также любого целого или вещественного типа.

При вводе переменных типа `Char` выполняется чтение одного символа. При вводе переменной типа `String` считывается вся строка.

При вводе числовых переменных все пробелы, символы табуляции и маркеры конца строк пропускаются. После выделения первого значащего символа, наоборот, любой из перечисленных символов служит признаком конца подстроки. Выделенная таким образом подстрока рассматривается как символьное представление числовой константы

соответствующего типа и преобразуется во внутреннее представление, а полученное значение присваивается переменной.

ReadLn (*файловая_переменная*, *список_ввода*) – обеспечивает ввод символов, строк и чисел. Эта процедура аналогична процедуре Read за исключением того, что после считывания последней в списке переменной происходит перевод указателя на начало новой строки.

Write (*файловая_переменная*, *список_вывода*) – обеспечивает вывод информации в текстовый файл. Список вывода – последовательность из одного или более выражений типа Char, String, а также любого целого или вещественного типа. При записи переменных в файл можно указывать форматы вывода.

WriteLn (*файловая_переменная*, *список_вывода*) – обеспечивает вывод информации в текстовый файл. Эта процедура аналогична процедуре Write за исключением того, что после вывода последней в списке переменной происходит перевод указателя на начало новой строки.

Функции для работы с текстовыми файлами

EOLN (*файловая_переменная*) – возвращает значение True, если достигнут маркер конца строки.

SeekEOLN (*файловая_переменная*) – пропускает все пробелы и знаки табуляции до маркера конца строки или до первого значащего символа и возвращает значение True, если маркер обнаружен.

SeekEOF (*файловая_переменная*) – пропускает все пробелы и знаки табуляции и маркеры конца строки до маркера конца файла или до первого значащего символа и возвращает значение True, если маркер обнаружен.

Порядок создания текстового файла

1. Присвоить имя файлу (Assign).
2. Открыть новый файл (Rewrite).
3. Записать компонент в файл (WriteLn).
4. Закрыть файл (Close).

Порядок использования текстового файла

1. Присвоить имя файлу (Assign).
2. Открыть существующий файл (Reset).

Таким образом, значение, на которое указывает указатель, т.е. собственно данные, размещенные в куче, обозначаются значком ^, который ставится сразу за указателем. Если за указателем нет значка ^, то имеется в виду *адрес,* по которому размещены данные.

Динамически размещенные данные можно использовать в любом месте программы, где это допустимо для констант и переменных соответствующего типа.

Динамическую память можно не только забирать из кучи, но и возвращать обратно. Для этого используется процедура Dispose. Например, операторы

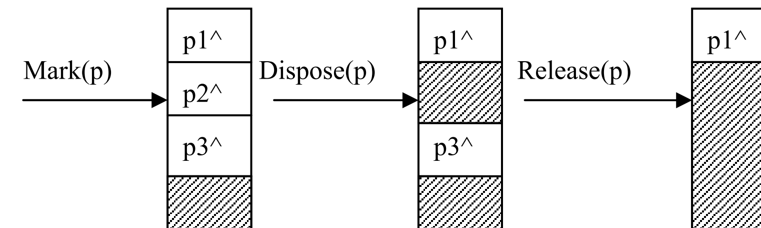
```
Dispose (r);
Dispose (i);
```

вернут в кучу 8 байт, которые ранее были выделены указателям i и R. Процедура Dispose(PTR) не изменяет значения указателя PTR, а лишь возвращает в кучу память, ранее связанную с этим указателем. Освободившийся указатель можно пометить зарезервированным словом Nil. Помечен ли какой-либо указатель или нет, можно проверить следующим образом:

```
Const
  p:^Real=NIL;
Begin
  If p=NIL then new(p);
  …
  Dispose(p);
  p:= NIL;
End.
```

Никакие другие операции сравнения над указателями не разрешены.

Чередование обращений к процедурам New и Dispose обычно приводит к «ячеистой» структуре памяти, изображенной на следующей схеме:



Состояние динамический памяти после действия различных процедур.

сток кучи от адреса, хранящегося в `PTR`, до конца кучи. Одновременно уничтожается список всех свободных фрагментов, которые, возможно, были созданы процедурами `Dispose` или `FreeMem`.

`SEG(x)` – функция возвращает значение типа `Word`, содержащее сегмент адреса указанного объекта. Здесь `x` – выражение любого типа или имя процедуры.

`SizeOf(x)` – функция возвращает длину в байтах внутреннего представления указанного объекта. Здесь `x` – имя переменной, функции или типа.

Выделение и освобождение динамической памяти

Память под любую динамически размещаемую переменную выделяется процедурой `New`. Параметром обращения к этой процедуре является типизированный указатель. В результате обращения указатель приобретает значение, соответствующее динамическому адресу, начиная с которого можно разместить данные, например:

```
Var
  i,j:^Integer;
  r: ^Real;
Begin
  New (i);
  ...
End.
```

После выполнения этого фрагмента указатель приобретет значение, которое перед этим имел указатель кучи `HeapPtr`, а сам `HeapPtr` увеличит свое значение на 2, так как длина внутреннего представления типа `Integer`, с которым связан указатель `i`, составляет 2 байта (на самом деле это не совсем так: память под любую переменную, выделяется порциями, кратными 8 байтам). Оператор

```
New(r);
```

вызовет еще раз смещение указателя `HeapPtr`, но теперь уже на 6 байт, потому что такова длина внутреннего представления типа `Real` Аналогичным образом выделяется память и для переменной любого другого типа. После того, как указатель приобрел некоторое значение, т.е. стал указывать на конкретный физический байт памяти, по этому адресу можно разместить любое значение соответствующего типа. Для этого сразу за указателем без каких – либо пробелов ставится значок ^, например:

```
i^:= 2; {В область памяти i помещено, значение 2}
```


3. прочитать компонент из файла (`ReadLn`).
4. Закрыть файл (`Close`).

Порядок корректировки текстового файла

Корректировка текстового файла заключается во внесении новых компонент в конец файла.

1. Присвоить имя файлу (`Assign`).
2. Открыть файл для дополнения (`Append`).
3. Записать компонент в файл (`WriteLn`).
4. Закрыть файл (`Close`).

Пример 1. Создать файл, компоненты которого являются строками. Распечатать все строки файла, длина которых более 20 символов.

```
Program Demo_1;
Var
  F:Text;
  St:String;
{создание текстового файла}
Procedure Sozd;
Begin
  Assign (F,'Stroki.dat');
  Rewrite(F);
  Writeln ('Введите строки файла');
  Writeln ('Признак окончания ввода – ***');
  While True do
  begin
    Readln(St);
    If St='***' then
    begin
      Close(F);
      Exit;
    end;
    WriteLn(F,St); {записали строку в файл}
  end;
End;
{обработка текстового файла}
Procedure Vivod;
Begin
  Assign (F,'Stroki.dat');
  Reset(F);
```

```
  While Not SeekEOF(F) do
  begin
    Readln(F,St); {прочитали строку из файла}
    If Length(St)>20 then
      WriteLn(St);
  end;
  Close(F);
End;
Begin {основная программа}
  Sozd;
  Vivod;
End.
```

Пример 2. Найти количество положительных элементов в целочисленном векторе A[1..5]. Результаты поместить в файл. Входной файл Input.txt содержит 5 чисел по одному в строке. Выходной файл Output.txt должен содержать единственное число – количество положительных элементов.

Пример входного файла Input.txt	Пример выходного файла Output.txt
5 -3 4 -1 1	2

```
Program Demo_2;
Var
  F:Text;
  A:array[1..5] of Integer;
  i,kol: Byte;
Begin
{считываем данные из файла}
  Assign (F,'Input.txt');
  Reset (F);
  For i:=1 to 5 do
    ReadLn(F,A[i]);
  Close(F);
{считаем количество положительных элементов}
  kol:=0;
  For i:=1 to 5 do
```


жится сегмент начала данных программы). Результат возвращается в слове типа Word.

FreeMem(нетипизированный_указатель,size) – процедура возвращает в кучу фрагмент динамической памяти, который ранее был зарезервирован за нетипизированным указателем. Здесь size — длина в байтах освобождаемого фрагмента.

GetMem(нетипизированный_указатель,size) – процедура резервирует за нетипизированным указателем фрагмент динамической памяти требуемого размера.

Mark(PTR) – процедура запоминает текущее значение указателя кучи HeapPtr. Здесь PTR — указатель любого типа, в котором будет возвращено текущее значение HeapPtr. Используется совместно с процедурой Release для освобождения части кучи.

MaxAvail – функция возвращает размер в байтах наибольшего непрерывного участка кучи. Результат имеет тип LongInt.

MemAvail – функция возвращает размер в байтах общего свободного пространства кучи. Результат имеет тип LongInt.

New(типизированный_указатель) – процедура резервирует фрагмент кучи для размещения переменной. Процедура New может вызываться как функция. В этом случае параметром обращения к ней является тип переменной, размещаемой в куче, а функция New возвращает значение типа указатель. Пример:

```
Type
  Pint=^Integer;
Var
  p: Pint;
Begin
  p:=New(Pint);
End.
```

QFS(x) – функция возвращает значение типа WORD, содержащее смещение адреса указанного объекта. Здесь x – выражение любого типа или имя процедуры.

PTR(SEG,OFS) – функция возвращает значение типа Pointer по заданному сегменту SEG и смещению OFS. Здесь SEG – выражение типа Word, содержащее сегмент; OFS — выражение типа WORD, содержащее смещение. Значение, возвращаемое функцией, совместимо с указателем любого типа.

RELEASE (PTR) – процедура освобождает участок кучи. Здесь PTR – указатель любого типа, в котором предварительно было сохранено процедурой Mark значение указателя кучи. Освобождается уча-

длину свободного блока или 0, если ниже адреса, содержащегося в `HeapPtr` нет свободных блоков. Ненормализованная длина определяется так: в старшем слове этого поля содержится количество свободных параграфов, а в младшем – количество свободных байт в диапазоне 0...15.

Сразу после загрузки программы указатели `HeapPtr` и `FreeList` содержат один и тот же адрес, который совпадает с началом кучи (этот адрес содержится в указателе `HeapOrg`*)*. При этом в первых 8 байтах кучи хранится запись, соответствующая типу `TFreeRec` (поле `Next` содержит адрес, совпадающий со значением `HeapEnd`*,* а поле `Size` – ноль, что служит дополнительным признаком отсутствия «ячеек» в динамической памяти). При работе с кучей указатели `HeapPtr` и `FreeList` будут иметь одинаковые значения до тех пор, пока в куче не образуется хотя бы один свободный блок ниже границы, содержащейся в указателе `HeapPtr`. Как только это произойдет, указатель `FreeList` станет ссылаться на начало этого блока, а в первых 8 байтах освобожденного участка памяти будет размещена запись `TFreeRec`. Используя `FreeList` как начало списка, программа пользователя всегда сможет просмотреть весь список свободных блоков и при необходимости модифицировать его.

Процедуры и функции для работы с динамической памятью

`ADDR(x)` –функция возвращает результат типа `Pointer`*,* в котором содержится адрес аргумента. Здесь `x` – любой объект программы (имя любой переменной, процедуры, функции). Возвращаемый адрес совместим с указателем любого типа. Аналогичный результат возвращает операция @.

`CSEG` – функция возвращает значение, хранящееся в регистре `CS` микропроцессора (в начале работы программы в регистре `CS` содержится сегмент начала кода программы). Результат возвращается в слове типа `Word`.

`Dispose(`типизированный_указатель`)` – процедура возвращает в кучу фрагмент динамической памяти, который ранее был зарезервирован за типизированным указателем. При повторном использовании процедуры применительно к уже освобожденному фрагменту возникает ошибка периода исполнения.

`DSEG` функция возвращает значение, хранящееся в регистре `DS` микропроцессора (в начале работы программы в регистре `DS` содер-


```
    If A[i]>0 then Inc(kol);
{записываем результат в файл}
  Assign (F,'Output.txt');
  Reset (F);
  WriteLn(F,kol);
  Close(F);
End.
```

Пример 3. Дана вещественная матрица `A`[1..5,1..5]. Построить вектор `B`[1..5], состоящий из сумм отрицательных элементов каждой строки матрицы `A`. Результаты поместить в файл. Входной файл `Input.txt` содержит 5 строк по 5 чисел в строке. Выходной файл `Output.txt` должен содержать элементы вектора B – по одному в строке.

Пример входного файла `Input.txt`	Пример выходного файла `Output.txt`
5.1 4.0 2.2 -1.0 6.3	-1.0
-3.1 6.1 -3.1 -2.2 6.8	-8.4
4.1 -4.5 2.2 -5.6 -3.2	-13.3
-1.1 1.0 2.1 -4.1 6.3	-5.2
1.1 4.7 2.5 -8.0 3.3	-8.0

```
Program Demo_3;
Var
  F:Text;
  A:array[1..5,1..5] of Real;
  B:array[1..5] of Real;
  i,j,sum: Byte;
Begin
{считываем данные из файла}
  Assign (F,'Input.txt');
  Reset (F);
  For i:=1 to 5 do
  begin
    For j:=1 to 5 do
      Read(F,A[i,j]);
    ReadLn(F);
  END;
  Close(F);
{считаем сумму отрицательных элементов в каждой строке}
  For i:=1 to 5 do
```

```
  begin
    sum:=0;
    For j:=1 to 5 do
      If A[i,j]<0 then sum:=sum+A[i,j];
    B[i]:=sum
  end;
{записываем результат в файл}
  Assign (F,'Output.txt');
  Reset (F);
  For i:=1 to 5 do
    WriteLn(F,B[i]);
  Close(F);
End.
```

Нетипизированные файлы

Формат описания:
1-ый способ:
Type
  имя_файлового_типа=**File**;
Var
  файловая_переменная: имя_файлового_типа;
2-ой способ:
Var
  файловая_переменная : **File**;

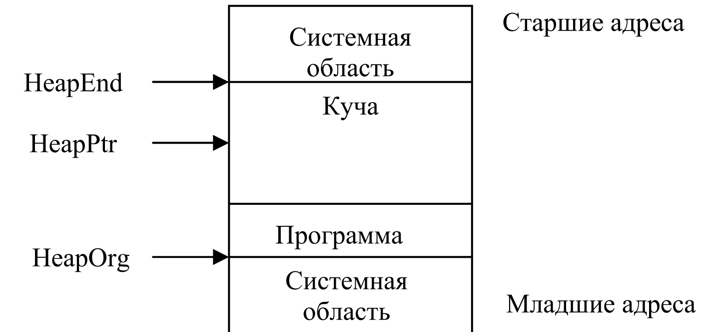
Нетипизированные файлы отличаются тем, что для них не указан тип компонент. Отсутствие типа делает их, с одной стороны, совместимыми с любыми другими файлами, а с другой – позволяет организовать высокоскоростной обмен данными между диском и памятью.

При инициализации нетипизированного файла процедурами Reset или Rewrite можно указать длину записи нетипизированного файла в байтах. Если длина записи не указана, принимается 128 байт.

Пример:

```
Var
  F:File;
Begin
  Assign(F,'MyFile.dat');
  Reset(F,512);
  ...
```


менной HeapOrg, конец — в переменной HeapEnd. Текущую границу незанятой динамической памяти указывает указатель HeapPtr.



Расположение кучи в памяти компьютера

Все операции с кучей выполняются под управлением особой подпрограммы, которая называется *администратором кучи.* Она автоматически пристыковывается к программе компоновщиком Turbo Pascal и ведет учет всех свободных фрагментов в куче.

Администратор кучи – это служебная подпрограмма, которая обеспечивает взаимодействие пользовательской программы с кучей. Администратор кучи обрабатывает запросы процедур работы с динамической памятью и изменяет значения указателей HeapPtr и FreeList. Указатель HeapPtr содержит адрес нижней границы свободной части кучи, а указатель FreeList — адрес описателя первого свободного блока. В модуле System указатель FreeList описан как Pointer, однако фактически он указывает на следующую структуру данных:

```
Type
  PFreeRec=^TFreeRec;
  TFreeRec=record
             Next: Pointer;
             Size : Pointer
           end;
```

Эта списочная структура предназначена для описания всех свободных блоков памяти, которые расположены ниже границы HeapPtr. Происхождение блоков связано с «ячеистой» структурой кучи. Поле Next в записи TFreeRec содержит адрес описателя следующего по списку свободного блока кучи или адрес, совпадающий с HeapEnd, если этот участок последний в списке. Поле Size содержит ненормализованную

Указатели такого рода называются *нетипизированными.* С их помощью удобно динамически размещать данные, структура и тип которых меняются в ходе работы программы.

Значениями указателей являются адреса переменных в памяти. Можно передавать значения только между указателями, связанными с одним и тем же типом данных. Если, например, объявлены указатели

```
Var
  p1,p2    :^Integer;
  p3       :^Real;
  pp       :Pointer;
```

то присваивание

```
p1:=p2;
```

вполне допустимо, в то время как

```
p1:= p3;
```

запрещено, поскольку `p1` и `p3` указывают на разные типы данных. Это ограничение, однако, не распространяется на нетипизированные указатели, поэтому можно записать

```
pp:=p3;
p1:=pp;
```

Обращение к значениям переменных заданного типа осуществляется следующим образом:

идентификатор^

Пример:

```
Var
  p:^Integer;
Begin
...
{будет распечатан шестнадцатеричный адрес переменной}
WriteLn(p);
{будет распечатано значение переменной по заданному адресу}
WriteLn(p^);
...
End;
```

Структура динамической памяти

Вся динамическая память в Turbo Pascal рассматривается как сплошной массив байтов, который называется *кучей*. Физически куча располагается в старших адресах сразу за областью памяти, которую занимает тело программы. Начало кучи хранится в стандартной пере-


```
End;
```

При работе с нетипизированными файлами могут применяться все процедуры и функции, доступные типизированным файлам, за исключением `Read` и `Write`, которые заменяются соответственно высокоскоростными процедурами `BlockRead` и `BlockWrite`. Формат описания процедур:

`BlockRead(`*файловая_переменная*`,`*буфер*`,count);`

`BlockRead(`*файловая_переменная*`,`*буфер*`,count,result);`

`BlockWrite(`*файловая_переменная*`,`*буфер*`,count);`

`BlockWrite(`*файловая_переменная*`,`*буфер*`,count,result);`

Буфер – это имя переменной, которая будет участвовать в обмене данными с диском. `Count` – количество блоков, которые нужно считать или записать. `Result` – количество блоков, которое было передано фактически.

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Динамическая память

Все переменные, объявленные в программе, размещаются в одной непрерывной области оперативной памяти, которая называется сегментом данных. Длина сегмента данных определяется архитектурой микропроцессоров 80x86 и составляет 65536 байт, что может вызвать затруднения при обработке больших массивов данных. С другой стороны, объем памяти персонального компьютера достаточен для успешного решения задач с большой размерностью данных. Выходом из положения может служить использование так называемой динамической памяти.

Динамическая *память* – это оперативная память персонального компьютера, предоставляемая программе при ее работе, за вычетом сегмента данных (64 Кбайт), стека (обычно 16 Кбайт) и собственно тела программы. По умолчанию размер динамической памяти определяется всей доступной памятью компьютера.

Динамическая память используется для обработки массивов данных большой размерности. Многие практические задачи трудно или невозможно решить без использования динамической памяти. Динамическое размещение данных означает использование динамической памяти непосредственно при работе программы. В отличие от этого, статическое размещение осуществляется компилятором Turbo Pascal в процессе компиляции программы. При динамическом размещении заранее не известны ни тип, ни количество размещаемых данных, к ним нельзя обращаться по именам, как к статическим переменным.

Адреса и указатели

Оперативная память представляет собой совокупность элементарных ячеек для хранения информации – байтов, каждый из которых имеет собственный номер. Эти номера называются *адресами,* они позволяют обращаться к любому байту памяти.

Для управления динамической памятью используются так называемые указатели. *Указатель* – это переменная, которая в качестве своего значения содержит адрес байта памяти. Адреса задаются совокупностью двух шестнадцатиразрядных слов, которые называются сегментом и смещением. *Сегмент* – это участок памяти, имеющий длину 65536 байт (64 Кбайт) и начинающийся с физического адреса, кратного 16 (т.е. 0, 16, 32, 48 и т.д.). Смещение указывает, сколько байт от нача-

ла сегмента необходимо пропустить, чтобы обратиться к нужному адресу.

Адресное пространство компьютера составляет 1 Мбайт. Для адресации в пределах 1 Мбайта нужно 20 двоичных разрядов, которые получаются из двух шестнадцатиразрядных слов (сегмента и смещения) следующим образом: содержимое сегмента смещается влево на 4 разряда, освободившиеся правые разряды заполняются нулями, результат складывается с содержимым смещения.

Физический адрес= Сегмент +Смещение

Фрагмент памяти в 16 байт называется *параграфом,* поэтому можно сказать, что сегмент адресует память с точностью до параграфа, а смещение – с точностью до байта. Каждому сегменту соответствует непрерывная и отдельно адресуемая область памяти. Сегменты могут следовать в памяти один за другим без промежутков или с некоторым интервалом, или перекрывать друг друга.

Таким образом, любой указатель представляет собой совокупность двух слов (данных типа `Word`*),* трактуемых как сегмент и смещение. С помощью указателей можно размещать в динамической памяти любые типы данных.

Объявление указателей

Как правило указатели связываются с некоторым типом данных. Такие указатели называются *типизированными*. Для объявления типизированного указателя используется значок ^, который помещается перед соответствующим типом.

Формат объявления типизированных указателей:

```
Var
  идентификатор:^тип_данных;
```

Пример:

```
Var
  P:^Integer; {объявлен указатель на данные целого типа}
  G:^Real;    {объявлен указатель на данные вещественного типа}
```

В Turbo Pascal можно объявлять указатель и не связывать его при этом с каким-либо конкретным типом данных. Для этого служит стандартный тип `Pointer`:

```
Var
  идентификатор: Pointer;
```