9. Больше о модульном программировании.

В данной главе описывается такой мощный механизм использования процедур и функций как рекурсия. Кроме того, даются примеры формирования меню пользователя, что также является важным фактором при структурном программировании. И, наконец, вводится понятие модуля. Рассматриваются также стандартные модули и их возможности.

9.1. Рекурсия (Recursion).

Говорится, что функция или процедура *рекурсивная,* если она вызывает себя. Для некоторых языков, таких, как языки функционального или логического программирования Lisp, Prolog, рекурсия является фундаментальным механизмом, Турбо-Паскаль также допускает реализацию принципов рекурсии.

В некоторых прикладных задачах рекурсия дает возможность записать решение в весьма привлекательной и компактной форме.

Главный принцип рекурсии заключается в следующем:

Предположим, решаемая проблема подпадает под какую-либо широкую категорию задач. Разбить проблему на несколько новых подзадач из той же категории. Написать подпрограмму, решающую задачи данной категории, которая будет состоять из одного или нескольких вызовов самой себя так, чтобы эти вызовы позволяли бы решать новые генерированные подзадачи.

Программа-пример: Вычисление факториала.

Классический пример использования концепта рекурсии демонстрируется на определении функции для вычисления факториала. При решении задачи подразумевается, что если мы знаем факториал от (n-1), то мы можем получить n! по формуле:

$$n!=n*(n-1)!$$

Т.е. мы свели исходную задачу (найти n!) к задаче того же типа (найти (n-1)!), но снизили значение «n». Если продолжать процесс,

то в конце дойдем до значения n=1. В программе этот случай рассматривается как предельный, и он служит условием завершения рекурсивных самовызовов. Кроме того, в главной программе используется цикл Repeat для реализации возможности ее повторного запуска. Программа будет запрашивать целое число до тех пор, пока пользователь не введет ноль или отрицательное число:

```
Program factorial;
Var n: Integer;

Function fact(i: Integer): LongInt;
Begin
    if i <= 1          { Условие прерывания рекурсии }
        Then fact := 1
        Else fact := i * fact(i - 1);    { Рекурсивный самовызов }
    End; { fact}

    BEGIN
        Repeat
            Write('Enter number from [0-16] (n<=0 - for the End):');
            Read(n);
            If n>0
                Then Writeln('Factorial: ', n,'! = ', fact(n),'.')
                Else WriteIn('Bye-bye!!!');
            Until n <= 0;
    END.
```

Программа-пример: Генерация чисел Фибоначчи.

Ниже приводится текст еще одной программы для генерации т.н. чисел Фибоначчи. Числами Фибоначчи называется последовательность чисел, начиная от единицы, каждое последующее из которых вычисляется как сумма двух предыдущих, т.е. 1 1 2 3 5 8 13 21 34 55 ...:

```
Program fibonachi;
Var n: Integer;
```

```
Function fib (i: Integer): LongInt;
Begin
    if i<l
     Then Writeln(' **** ')
     Else
      If (i= 1) Or (i=2)
          Then fib := 1    {Условие прерывания рекурсии }
          Else fib:= fib(i-l) + fib(i-2);
End; { fib }

BEGIN
 Repeat
    Write('Enter number   (n<=0 - for the End):');
    Read(n);
    If n>0
          Then Writeln('fibonachi number ', n,' = ', fib(n),'.')
          ElseWriteln('Bye-bye!!!');
    Until n <= 0;
END.
```

Программа-пример: Быстрая сортировка.

Теперь рассмотрим пример использования рекурсивной процедуры для реализации алгоритма т.н. «быстрой сортировки» (Quicksort)-механизма для сортировки массива элементов по возрастанию. В этом случае «категорией» является сортировка. Разобьем задачу на подзадачи следующим образом. Для удобства рассмотрим конкретный массив:

12 5 66 13 8 35 29 88

Теперь выбираем первый элемент, т.е. 12, как отправную точку. Затем будем формировать два новых массива — *Smaller* и *Larger,* состоящих из элементов, которые соответственно меньше и больше первого отправного элемента. В результате получим:

Smaller 5 8
Larger 66 13 35 29 88

Теперь мы столкнулись с двумя новыми задачами «той же категории», что и исходная задача, т.е. категории сортировки — сортировать список *Smaller* и список *Larger*. В результате должны получить:

Smaller 5 8
Larger 13 29 35 66 88

Сейчас можно соединить три пункта — массив *Smaller*, отправной элемент и массив *Larger*:

5 8 13 29 35 66 88

Таким образом, исходный массив отсортирован. Ключевым моментом здесь было то, что все элементы массива *Smaller* меньше, чем 12, а все элементы массива *Larger* больше, чем 12, так что мы получаем правильный порядок только соединением этих трех массивов.

Ниже приводится код программы с использованием процедуры *Quicksort*. Так как структурный тип — массив объясняется в двенадцатой главе, здесь отметим, что это совокупность однотипных элементов, к которым можно обращаться по индексу. Если программа все же будет сложной для восприятия, к ней можно вернуться уже после того, как вы освоите структурные типы данных.

```
Program recursionSort;
Type sortarray = Array[0..100] Of Integer;
Var x: sortarray;
    n, i, temp: Integer;

 Procedure quicksort (Var x: sortarray; n: Integer);
 Var smaller, larger: sortarray;
    i, small J, largeJ, compare Value: Integer;
```

```
Begin
 If(n <= 1) Then Exit;   { Условие прерывания рекурсии }
 { Отправной элемент }
 compare Value := x[1];
 smallJ := 0;
 largeJ ;= 0;
 { Формирование массивов smaller и larger }
 For i := 2 To n Do
     If(x[i] <= compareValue)
     Then
     Begin
         smallJ := smallJ+ 1;
         smaller[smallJ] := x[i];
     End {Then}
     Else
         Begin
         largeJ := largeJ + 1;
         larger[largeJ] := x[i];
     End; {Else}
     { Рекурсивные самовызовы }
quicksort(smaller, smallJ);
quicksort(larger, largeJ);
{Соединение отсортированных массивов smaller и larger}
  For i := 1 To smallJ Do x[i] := smaller[i];
 x[smallJ+ 1] := compare Value;
 For i := 1 To largeJ Do x[smallJ + 1 + i] := larger[i];
 End; { quicksort }

 BEGIN
    Writeln('Enter array:   / -1 for the end');
    n:=0;
    Repeat
        Inc(n);
        Write('x[', n : 2, ']=');
       Read(x[n]);
       Until x[n]=-l;
       Dec(n);
```

```
        Writeln;
        Writeln('n=', n : 2);
        quicksort(x, n);
        Writeln; Writeln('Sorted array:');
        For i := 1 To n Do Writeln('x[', i: 2, ']=', x[i],'.');
END.
```

Процедура следует вышеописанному методу сортировки, т.е. формирует два подмассива, сортирует их и затем соединяет вместе результаты. Отметим некоторые моменты:

Каждая рекурсивная подпрограмма должна включать в себя специальный случай, состояние задачи, которое будет выполняться непосредственно, а не с помощью рекурсивных вызовов. Это можно назвать граничным, предельным условием завершения подпрограммы, которое предотвращает зацикливание и гарантирует ее конечность. В приведенном примере такое граничное условие содержится в строке.

```
If(n<= 1) Then Exit;
```

т.е. когда новый массив, полученный в результате очередного расщепления, будет состоять из одного элемента, дальнейшая обработка прерывается и рекурсивные вызовы процедуры прекращаются.

С другой стороны, в процессе составления программы вы не должны задумываться о непрерывной последовательности вызовов, которые будут предприниматься в действительности. Думать надо только о том, как преобразовать вашу проблему в одну или несколько маленьких подзадач.

9.2. Использование Меню (Menu) для организации пользовательской программы.

Слово *Меню* предполагает выбор. Главная программа может предложить несколько альтернатив, каждая из которых выполняет какую-нибудь подзадачу. Таким образом, можно обеспечить решение нескольких задач в пределах одной программы.

Пусть требуется организовать основную программу, которая позволяла бы продемонстрировать все задачи, решенные студентом. Естественно, можно каждую задачу оформить как отдельную программу, но можно организовать более удобный интерфейс с помощью меню.

Пусть студенту заданы следующие задачи:

1) Решить линейное уравнение;
2) Решить квадратное уравнение;
3) С клавиатуры задается последовательность чисел, признаком окончания ввода служит число 0. Надо подсчитать количество положительных и отрицательных чисел,
4) Условие третьей задачи только требует определить минимальное и максимальное из введенных чисел.

Каждую из этих программ можно написать, откорректировать и отладить самостоятельно, а затем использовать в главной программе, которая будет работать следующим образом:

На экране появляется меню, с перечислением всех задач, которые могут быть выполнены в пределах данной программы. Каждая задача активизируется только при нажатии определенной клавиши (или последовательности клавиш, или мыши). Для приведенного примера такое меню может иметь вид:

Зачетные задачи студента ИМЯ, ФАМИЛИЯ, ГРУППА, КУРС

1. Решение линейного уравнения;
2. Решение квадратного уравнения;
3. Подсчет положительных и отрицательных чисел;
4. Нахождение MIN и MAX из последовательности чисел;
5. Выход

Выберите вариант из 1 — 5

Этот интерфейс очень легко организовать с помощью оператора «Writeln». Дальнейшее выполнение программы же можно реализовать с помощью дополнительной переменной, скажем *key* и

оператора выбора «Case» и цикла «Repeat» (для возможности повторного запуска главной программы, см. предыдущий раздел).

Пусть отдельные программы уже готовы и оформлены в виде процедур соответственно:

1) Procedure LinearEq(a,b:Real);
2) Procedure SqrEq(a,b,c:Real);
3) Procedure CalcNumbs;
4) Procedure FindMinMax;

Тогда при нажатии клавиши из меню выбора можно обработать его значение в операторе «Case» и каждый пункт меню реализовать с помощью вызова соответствующей процедуры.

Если принять, что перечисленные выше процедуры существуют, тогда программа, реализующая поставленую проблему, может иметь вид:

```
{*************************************************}
{*          Main Program                         *}
{*************************************************}

Program First_Course;
Uses Crt;
Var i: Integer;
    a,b,c:Real;
    key:Char;
Procedure LinearEq(a,b:Real);
Var x:Real;
Begin
    { Реализация тела процедуры LinearEq }
End; { LinearEq}

Procedure SqrEq(a,b,c:Real);
Var x,x1,x2,D:Real;
Begin
    { Реализация тела процедуры SqrEq }
End; { SqrEq }
```

```
Procedure CalcNumbs;
Var number,kol_pos,kol_neg:Real;
Begin
    { Реализация тела процедуры CalcNumbs }
End; { CalcNumbs }

Procedure FindMinMax;
Var number,min,max:Real;
Begin
    { Реализация тела процедуры FindMinMax }
End; { FindMinMax }

BEGIN
Repeat
    ClrScr;
    For i:=1 to 5 Do Writeln;

    Writeln(':10,'Зачетные задачи студента ИМЯ, ФАМИЛИЯ,
ГРУППА, КУРС ');
    Writeln;
    Writeln(' ':20,'1. Решение линейного уравнения;');
    Writeln(' ':20,'2. Решение квадратного уравнения;');
    Writeln(' ':20,'3. Подсчет положительных и отрицательных
чисел;');
    Writeln(' ':20,'4. Найти MIN и MAX из последовательности
чисел;');
    Writeln(':20;5. Выход');
    Writeln;
    Writeln(': 10,'Выберите вариант из 1 — 5');

    key:=readKey;
    Case key of
    '1': Begin
    Write(' Введите коэффициенты для линейного уравнения ф и
    и:');
        Read(a,b);
```

```
        LinearEq(a,b);
        End; {'1'}
    '2': Begin
    Write(' Введите коэффициенты для квадратного уравнения a, b и
    c:');
        Read(a,b,c);
        SqrEq(a,b,c);
        End; {'2'}
    '3': CalcNumbs;
        '4': FindMinMax;
        '5': Halt;
    Else Begin WriteIn('Type number only from the RANGE: 1 - 5 !!!');
    Delay(1000) End
End {Case}

Until FALSE
END.
{*************************************************}
{*          End of Program                       *}
{*************************************************}
```

9.3. Опережающее описание.

По структуре Турбо-Паскалевской программы все элементы, использованные на каждом этапе, должны быть предварительно объявлены (исключением является тип-указатель, см. ниже). Следовательно, описания процедур и функций также должны предшествовать главному коду программы, где они будут вызываться. Это несколько затрудняет сохранить читабельный вид текста, было бы удобнее, если бы сначала шел код главной программы, а затем – описания процедур и функций. Кроме того, возникает затруднение, если две или больше процедуры взаимовызывают друг друга, и становится непонятным, какая же из них должна быть описана первой. В Турбо-Паскале есть специальная возможность, допускающая упоминание процедур и функций в программном коде, тогда как их описание располагается в конце главного кода. Эта возможность осуществляется с помощью

ключевого слова «Forward» и строки т.н. *опережающего* объявления:

```
Procedure имя(список — параметров);
Forward;
```

Т.е. описывается только заголовок с формальными параметрами, а тело заменяется ключевым словом «Forward». Описание тела теперь можно расположить в любом месте программного кода. Рассмотрим формальный пример взаимовызова двух *процедур first* и *second.* и их объявления:

```
Procedure first(i:Integer);
Forward;
Procedure second(r:Real);
Begin
...
first(i);
...
End;

Procedure first;
Begin
...
second(r);
...
End;

BEGIN { Главная программа }
...
 first(x);
...
END.
```

Отметим, что при описании тела процедуры *first* перечень формальных параметров заново не задается.

9.4. Основные стандартные модули.

Все стандартные процедуры и функции, которые мы рассматривали до сих пор, доступны пользователю только потому, что «кто-то» их написал за нас. Все эти подпрограммы собраны вместе в одной «библиотеке» – модуле под именем System.tpu. Этот модуль автоматически загружается в оперативную память вместе с системой редактора Turbo Pascal и, следовательно, все процедуры, функции, константы и переменные становятся доступны пользователю.

Кроме системного модуля, к Турбо-Паскалевскому редактору прилагаются другие «библиотеки» процедур и функций, которыми, при желании, может воспользоваться программист. Но для этого компилятору нужно указать, в каком именно модуле надо искать те подпрограммы, которые вызываются в программном коде. Синтаксически это можно осуществить следующей инструкцией, которая должна присутствовать перед всеми блоками объявлений:

```
Uses имя-модуля;
```

Здесь мы приведем описание основных стандартных модулей и перечень большинства содержащихся в них процедур и функций, некоторые примеры их применения, а подпрограммы, использующие сложные структурированные элементы, будут рассмотрены в последней главе, оснащенной примерами.

Основные стандартные модули:

```
Crt.tpu
Graph.tpu
Dos.tpu
Printer.tpu
```

Первые два модуля предназначены для дополнительного управления ввода/вывода, а также управления экраном, соответственно, в *текстовом* (Crt) и *графическом* (Graph) режимах. Модуль Dos дает возможность использовать команды и возможности операционной системы, а последний модуль Printer позволяет вывод выходных данных на печать, а именно, в модуле

определена переменная *Lst*, которая связывается с портом принтера. Задав оператор

```
Uses Printer;
...
Writeln(Lst,' Текст для вывода на принтер.'),
```

можно вывести соответствующее сообщение на печать. Мы могли реализовать эту возможность следующим образом:

```
Var Lst:Text;
...
Assign(Lst, LPT1);
ReWrite(Lst);
...
Writeln(Lst,' Текст для вывода на принтер.');
```

Здесь мы приведем более детальные комментарии только к процедурам и функциям из модулей Crt и Graph, которые дают дополнительные возможности для работы с экраном.

Экран дисплея представляет собой электронно-лучевую трубку, как и кинескоп телевизора. Электронный луч обегает экран несколько десятков раз за минуту, с помощью чего и создается изображение. Весь экран можно рассматривать как совокупность светящихся точек различной яркости и цвета, называемых *пикселями* (pixel, от picture cell), каждым из которых можно управлять программными средствами, изменяя энергию электронного луча.

Во время управления экраном связывающим звеном между программой и монитором, как техническим элементом, служит т.н. *видеопамять*, которая служит местом временного хранения передаваемой со стороны программы информации, что значительно экономит время выполнения программы. Электронные компоненты схемы управления монитором, программируемые каналы ввода и вывода, видеопамять и другие элементы располагаются на отдельной плате, которая называется *адаптером*. Перечислим основные типы адаптеров:

MDA (Monocrom Display Addapter) — монохромный

CGA (Colour Graphic Addapter)
EGA (Enhanced Graphic Addapter)
VGA (Video Graphic Addapter)
S-VGA (Super Video Graphic Addapter)

Количество точек по горизонтали и вертикали называется разрешением монитора и зависит от типа его адаптера. Каждый монитор имеет максимально допустимое разрешение, но позволяет работать в режимах с более низким разрешением. Соответственно, монитор может работать в различных *режимах.* В таблице 9.1 дается классификационная схема для адаптеров различных типов.

Таблица 9.1

Тип адаптера	Допустимые разрешения
CGA	640x200
EGA	640x350
VGA	640x480
S-VGA	1024x768

Как в текстовом, так и графическом режимах на экране присутствует т.н. *курсор,* который в каждый момент имеет определенное текущее состояние. При команде о выводе данных информация (текстовая или графическая) начинает поступать на экран именно с текущей позиции курсора. Эту позицию можно контролировать различными командами, которые будут рассмотрены ниже.

В текстовом режиме экран рассматривается как совокупность клеточек, ячеек, размер которых определяется, с одной стороны, типом монитора, а, с другой, – текущим режимом экрана. Например, при разрешении экрана 640x200 и размера клеточки 8x8 пикселей всего по горизонтали разместятся 80, а по вертикали – 25 клеточек. Каждая ячейка имеет собственные координаты, определяющие ее позицию на экране. Если предположить, что по горизонтали у нас направлена ось *X,* а по вертикали вниз — ось *Y,* то координатами левой верхней ячейки будут (1,1). Соответственно, координатами ячейки, расположенной в десятой строке и тридцатом столбце, будут

(30,10). В текстовом режиме на экран можно вывести символы из ASCII—таблицы.

Таким образом, в текстовом режиме можно управлять экраном с точностью до ячейки.

В графическом режиме можно управлять экраном с точностью до пикселя. Каждый пиксель имеет собственные координаты. Направление координатных осей то же самое, что и в текстовом режиме, но отсчет начинается от нуля, т.е. координатами левого верхнего пикселя будут (0.0). Координаты пикселя в правом нижнем углу в этом случае зависят от текущего разрешения дисплея. Естественно, чем больше разрешение, тем четче изображение на экране. В свою очередь, максимальное разрешение экрана определяется размером видеопамяти соответствующего монитора.

Управление экраном в графическом режиме осуществляется с помощью вспомогательной программы, называемой *драйвером,* которая обеспечивает связь с техническими средствами. Каждый тип монитора управляется собственным графическим драйвером. Файлы с данными программы имеют расширение «BGI» (Borland Graphics Interface — графический интерфейс фирмы Borland). Например, самые распространенные: CGA.bgi — для CGA адаптера, EGAVGA.bgi — для EGA и VGA адаптера, и др.

9.4.1. Модуль Crt.tpu (для работы в текстовом режиме).

Процедуры:

TextColor(colour: Byte)

Устанавливает номер текущего цвета для вывода текста. Мы как бы выбираем цветной карандаш, результат этого выбора проявляется при первом же выводе информации.

TextBackGround(colour:Byte)

Устанавливает номер текущего цвета для фона выводимого текста.

ClrScr — (сокращенное от Clear Screen)

Очищает экран дисплея и заполняет его текущим цветом фона.

ClrEoL — (сокращенное от Clear End of Line)

Очищает правую от курсора часть строки.

InsLine — (сокращенное от Insert Line)

Вставляет пустую строку, при этом курсор с текущей строки, и вся нижняя часть экрана смещается вниз.

DelLine — (сокращенное от Delete Line)

Удаляет текущую строку, нижняя часть экрана смещается вверх на одну строку.

TextMode(mode:ShortInt)

Задает новый текстовой режим монитора. При этом курсор устанавливается в левом верхнем углу экрана.

Goto(x,y:Byte)

Устанавливает курсор в позицию с координатами *(x,y)*, т.е. на ячейку, расположенную на строке *y* в столбце *x*. Это означает, что весь последующий вывод будет осуществляться именно с данной позиции.

Window(xl ,yl,x2,y2:Byte)

Задает отдельное окно для вывода, при этом, *x1* и *yl* — координаты левого верхнего угла, а *x2* и *y2* — координаты правого нижнего угла активного окна. Должны соблюдаться условия: *x1<x2* и *y1<y2*. После вызова данной процедуры весь отсчет координат осуществляется относительно нового окна, т.е. ячейка *(x1,x2)* теперь имеет координаты (1,1).

Sound(hz:Word)

Генерирует звук с частотой *hz*.

NoSound

Отключает звук.

Delay(mS:Word)

Задерживает изображение на экране на *mS* миллисекунд.

Следующие три процедуры устанавливают различную яркость для выводимого текста: **Low Video** (пониженную), **Norm Video** (нормальную) и **High Video** (яркую).

Функции:

KeyPressed : Boolean —

Логическая функция. Когда в программном коде вызывается эта функция, ее значение остается *False* до тех пор, пока пользователь не нажмет какую-нибудь клавишу, после нажатия клавиши она принимает значение *True*. Преимущественно используется в условных выражениях при организации циклов.

WhereX : Byte -

Возвращает *x* координату текущей позиции курсора.

WhereY : Byte 2

Возвращает *y* координату текущей позиции курсора.

ReadKey : Char 2

Используется для ввода, считывания символа с клавиатуры. Формат вызова, например, может иметь вид;

```
c:=ReadKey;
```

При этом программа переходит в состояние ожидания, как и в случае оператора «Read». Как только пользователь нажмет на клавишу, значение, соответствующее данной клавише, присваивается символьной переменной *c* (символ на экране не отображается),

В модуле Crt также определены константы для представления цветов:

Черный – Black=0;
Синий – Blue=l;
Зеленый – Green=2;
Голубой – Cyan=3;
Красный – Red=4;
Фиолетовый – Magenta=5;
Коричневый – Brown=6;
Светло-серый – LightGrey=7;
Темно-серый – Grey=8;
Ярко-синий – LightBlue=9;
Ярко-зеленый – LightGreen=10;
Ярко-голубой – LightCyan=ll;
Розовый – LightRed=12;
Малиновый – LightMagenta= 13;
Желтый – Yellow=14;
Белый – White=15;
Мерцание символа – Blink=128;

По умолчанию цветом фона установлен черный *(Black)*, а цветом текста — серый *(Grey)*.

9.4.2. Модуль Graph.tpu (для работы в графическом режиме).

По умолчанию программный вывод информации на экран осуществляется в текстовом режиме. Чтобы иметь возможность работать в графическом режиме, его нужно специально *инициировать*. В этом разделе мы рассмотрим процедуры и функции, которые позволяют работать в графическом режиме. Это включает — вывод графической информации на экран, возможность получить и изменить значения параметров режима и др.

Инициация графического режима и некоторые характеристики.

InitGraph(Var driver, mode:Integer; path:String);

Инициирует графический режим работы монитора. Все процедуры и функции, а также константы и переменные, описанные в модуле «Graph.tpu», смогут быть задействованы в программном коде только после вызова процедуры инициации.

Параметры:

driver — определяет тип драйвера. Для данного параметра в модуле объявлены специальные константы. Вот некоторые из них:

Detect=0;
CGA=1;
EGA=3;
VGA=9; и др.

Из этих величин *Detect* имеет специальное назначение, а именно, если *driver-Detect* (=0), происходит автоматическое определение типа драйвера, а режим устанавливается на максимально допустимое разрешение для данного типа драйвера. Во всех остальных случаях величину режима нужно задавать вручную, т.е. определяется пользователем.

mode — задает графический режим из допустимых для данного типа драйвера.

Для режима определены константы, среди них:

```
EGALo=0;    {640 x 200, 16 цветов, 4 страницы}
EGAHi=1;    {640 x 350, 16 цветов, 2 страницы}
VGALo=0;    {640 x 200, 16 цветов, 4 страницы}
VGAMed=1;   {640 x 350, 16 цветов, 2 страницы}
VGAHi=2;    {640 x 480, 16 цветов, 1 страница }
```

Здесь упоминается графическая страница, это одно графическое изображение в полный экран. В зависимости от размера

видеопамяти и разрешения (на большее разрешение уходит больше памяти) можно обрабатывать несколько страниц одновременно, т.е. хранить их всех в видеопамяти.
path — путь к каталогу, в котором располагается файл драйвера.

CloseGraph;

Закрывает графическую систему и возвращает монитор в текстовой режим работы. Если этого не сделать программно, после завершения выполнения вашей программы система сама возвратится в текстовой режим.

GraphResult: Integer;

Возвращает код ошибки для последней графической операции. Определены следующие константы:

```
grOk=0;                                    { Ошибок нет }
grInitGraph=-l;
            { Графический режим не был инициирован }
grNotDetected=-2;          {Не определен тип драйвера }
grFileNotFind=-3;
            { Не найден файл с графическим драйвером }
grInvalidDriver=-4      { Указан неверный тип драйвера }
grNoLoadMem=-5;
             { Не хватает памяти для загрузки драйвера }
grNoScanMem=-6;
            { Не хватает памяти для просмотра областей }
grNoFloodMem=-7;
             { Не хватает памяти для закраски областей }
grFontNotFound=-8;
                        { Не найден файл со шрифтом }
grNoFontMem=-9;
                 { Не хватает памяти для загрузки шрифта }
grInvalidMode=-10;
                  { Указан неверный графический режим }
grError=-l 1;                          { Общая ошибка }
grIOError=-12;                    { Ошибка ввода/вывода }
grInvalidFont=-13;     { Указан неверный формат шрифта }
```

grInvalidFontNum=-14;{ Указан неверный номер шрифта }

GraphErrorMsg(errorCode: Integer): string;

Возвращает текстовое сообщение об ошибке в графической операции для заданного кода ошибки *errorCode*.

RestoreCrtMode;

Восстанавливает текстовой режим монитора, который был до инициализации графики.

GetGraphMode: Integer;

Возвращает текущий графический режим.

GetDriverName: string;

Возвращает имя текущего драйвера.

GetModeName(modeNumber: Integer): string;

Возвращает имя графического режима, который соответствует аргументу *modeNumber*.

GetMaxMode: Integer;

Возвращает номер максимального режима для текущего загруженного драйвера.

DetectGraph(Var graphDriver, graphMode: Integer);

Проверяет аппаратуру, определяет и выдает текущий графический драйвер и режим в переменных *graphDriver* и *graphMode*.

GetModeRange(graphDriver:Integer; Var loMode, hiMode:Integer);

Возвращает минимальный *(loMode)* и максимальный *(hiMode)* графические режимы для драйвера *graphDriver*.

SetGraphMode(mode: Integer);

Переводит систему в графический режим под кодом *mode* и очищает экран.

GetMaxX: Integer;

Возвращает максимальное разрешение по горизонтали (*X*) для текущего графического драйвера и режима.

GetMaxY: Integer;

Возвращает максимальное разрешение по вертикали (Y) для текущего графического драйвера и режима.

GetX: Integer;

Возвращает *x*-координату текущей позиции курсора.

GetY: Integer;

Возвращает *y*-координату текущей позиции курсора.

SetViewPort(xl, yl, x2, y2:Integer; clip:Boolean);

Устанавливает текущее окно для графического вывода.

xl, y1, x2, y2 — координаты окна.

clip — устанавливает режим для отсечения не поместившихся в окне графических элементов: при *clip=False* — границы окна игнорируются, в противном случае *(clip=True)* — лишняя часть отсекается.

GetViewSettings(Var viewPort: ViewPortType);

Выдает параметры текущего окна и отсечения в переменной типа *ViewPort:*

```
ViewPortType = Record
    xl, yl, x2, y2: Integer
    clip: Boolean
    End;
```

(Для идентификации полей записи смотрите параметры процедуры Set View Port.)

MoveRel(dX, dY: Integer);
Перемещает курсор на заданное расстояние от его текущей позиции. Относительное смещение определяется по горизонтали — величиной *dX*, а по вертикали — величиной *dY*.

MoveTo(x, y: Integer);
Перемещает курсор в точку с координатами *(x, y)*.

ClearDevice;
Очищает экран и устанавливает курсор в начало отсчета координат.

ClearViewPort;
Очищает текущее окно.

GetAspectRatio(Var xAsp, yAsp: Word);
Возвращает относительное разрешение, т.е. соотношение сторон экрана в пикселях, которое можно вычислить с помощью переменных *xAsp* и *yAsp*.

SetAspectRatio(xAsp, yAsp: Word): Word;
Устанавливает новое относительное разрешение — новое соотношение сторон экрана.

SetActivePage(page: Word);
Устанавливает активную страницу для графического вывода.

SetVisualPage(page: Word);
Устанавливает номер видимой графической страницы.

Линии и фигуры, шаблоны линий.
Линии
GetPixel(x,y: Integer): Word;
Возвращает цвет пикселя с координатами *(x,y)*.

PutPixel(x,y: Integer; pixel: Word);
Ставит точку цвета *pixel* в *(x,y)*.

Line(xl, yl, x2, y2: Integer);

Проводит линию от точки с координатами *(x1, y1) до* точки *(x2, y2).*

LineTo(x, y: Integer);

Проводит линию от текущей позиции курсора до точки с координатами *(x,y).*

LineRel(dX, dY: Integer);

Проводит линию от текущей позиции курсора до точки, лежащей на заданном аргументами *dX* и *dY* расстоянии.

SetLineStyle(lineStyle:Word; pattern:Word; thickness:Word);

Устанавливает текущий тип *(lineStyle),* шаблон *(pattern)* и толщину *(thickness)* линии.

Для типа линии определены следующие константы:

SolidLn=0; — сплошная линия;
DottedLn=l; — точечная линия;
SolidLn=0; — штрих—пунктир;
SolidLn=0; — пунктир;
UserBitLn=0; — определяется пользователем.

Шаблон принимается в расчет, только если *UserBitLn=0.* Это параметр типа Word, занимающий 2 байта памяти, с помощью которых пользователь может задать свой собственный тип линии шаблоном длиной в 16 пикселей, который будет периодически повторяться по всей длине выводимой линии.

Для *толщины* определены две константы:

NormWidth=l;

Устанавливает толщину линии в один пиксель,
ThickWidth=l;

Устанавливает толщину линии в три пикселя;

SetWriteMode(writeMode: Integer);

Устанавливает режим вывода для линий, рисуемых с помощью процедур *DrawPoly, Line, LineRel, LineTo* и *Rectangle.* При выводе

изображений может быть установлен один из двух режимов накладывания линий: либо обычным способом *(режим=0)*, либо по принципу исключающее ИЛИ, т.е. если при выводе в определенных точках экрана изображение там уже присутствовало, светимость этих точек инвертируется *(режим=1)*. Для параметра режима определены две константы.

CopyPut=0,

XorPut=l,

GetLineSettings(Var lineInfo: LineSettingsType);

Выдает текущий тип, шаблон и толщину линии, установленных процедурой *SetLineStyle*

Фигуры

Rectangle(xl, yl, x2, y2: Integer);

Чертит прямоугольник текущим цветом и типом линии. Задаются координаты левого верхнего *(x1,y1)* и правого нижнего *(x2,y2)* углов,

Circle(x,y: Integer; radius: Word);

Чертит окружность с радиусом *radius*, используя точку *(x,y)* как центр.

Ellipse(x, y: Integer; stAngle, endAngle: Word; xRadius, yRadius: Word);

Чертит эллиптическую дугу с центром в точке *(x,y)*, от начального угла *stAngle* до конечного *endAngle* против часовой стрелки, используя *xRadius*, и *yRadius* как горизонтальную и вертикальную оси, соответственно.

Arc(x,y: Integer; stAngle, endAngle, radius: Word);

Чертит дугу с радиусом *radius* от начального угла *stAngle* до конечного *endAngle* против часовой стрелки, используя точку (x,y) как центр.

Drawpoly(numPoints: Word; Var PolyPoints);

Чертит многоугольник текущим цветом и типом линии. Количество угловых точек определяется с помощью аргумента *(numPoints-1)*, а их координаты задаются массивом элементов типа записи *PointType*:

```
Type PointType = Record
                   x,y: Word
                 End;
```

GetArcCoords(Var arcCoords: ArcCoordsType);

В переменной *arcCoords* выдает координаты трех точек (центра, начала и конца) дуги, проведенной с помощью процедуры *Arc*.

```
Type  ArcCoordsType = Record
{ Центр }                    x,y : Integer;
{ Начало }           xStart,yStart: Integer;
{ Конец }             xEnd,yEnd : Integer;
                End;
```

Управление цветом и шаблоны для штриховки.

Цвета

EGA адаптер может генерировать 64 цвета, каждый из которых имеет собственный код (нумерация начинается с нуля). Допустимые для одновременного использования цвета определены в т.н. *палитре.* Набор активных цветов в палитре можно контролировать с помощью соответствующих процедур. EGA—палитра содержит 16 цветов. В модуле определена константа:

```
Const maxColor=15;
```

GetColor: Word;

Возвращает текущий активный цвет.

GetBkColor: Word;

Возвращает текущий цвет фона.

SetColor(color: Word);

Устанавливает активный цвет под номером *color*, которым будет осуществляться рисование.

SetBkColor(color: Word);

Устанавливает цвет фона под номером *color*.

GetDefaultPalette(Var palette: PaletteType);

Выводит аппаратную палитру в структурированной переменной—записи *PaletteType:*

```
Type  PaletteType = Record
                      size:Word;
                      colors:Array [O..MaxColors] of ShortInt
                    End;
```

Здесь *size* — количество цветов в палитре; а *colors* — коды цветов, входящих в палитру.

SetAllPalette(Var palette);

Изменяет цвет палитры.

SetPalette(colorOld: Word; colorNew: Shortint);

Изменяет цвет палитры с кодом *colorOld* цветом с кодом *colorNew*.

GetMaxColor: Word;

Возвращает максимальный номер цвета, который можно задать в процедуре *SetColor*.

GetPaletteSize: Integer;

Возвращает номер таблицы палитры.

GetPalette(Var palette: PaletteType);

Возвращает текущую палитру и ее размер.

Штриховка

SetFillStyle(pattern: Word; color: Word);

Устанавливает шаблон штриховки *(pattern)* и цвет *(color)*, один из стандартных типов. Тип штриховки определяется повторяющимся узором, которым можно заполнить замкнутую область на экране. В модуле определены константы для задания стандартных шаблонов:

```
EmptyFill=0;                    { без узора }
SolidFill=1;                    { сплошная}
LineFill=2;                     {линиями }
LtSlashFill=3;                  { узор с /// }
SlashFill=4;            { узор с утолщенными /// }
BkSlashFiII=5;                  {узор \\\}
LtBkSlashFill=6         { узор с утолщенными \\\}
HatchFill=7;                    { узор с + + + }
XHatchFill=8;                   { узор с - - - }
InterLeaveFill=9;          { прямоугольная сетка}
WideDotFiII=10;         { узор с редкими точками }
CloseDotFill=11         { узор с частыми точками }
UserFill=12;  { узор определяется пользователем }
```

SetFillPattern(pattern: FillPatternType; Color: Word);

Устанавливает шаблон штриховки *(pattern)* и цвета *color*, определенный пользователем.

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

Переменная типа *FillPatternType* определяет узор для области в 8x8 пикселей. Каждый байт в битовой интерпретации представляет одну строку из восьми последовательных пикселей, т.е. двоичная интерпретация байта определяет светимость восьми пикселей одного ряда, что в итоге создает квадратный шаблон. Ниже приведен пример шаблона «колокольчик», определенного пользователем. Значения байтов даются также в шестнадцатеричном представлении, так как будет легче воспроизвести картину из двоичных цифр:

Таблица 9.2

Элементы массива		Рисунок шаблона							
38	$26	0	0	1	0	0	1	1	0
231	$E7	1	1	1	0	0	1	1	1
105	$59	0	1	1	0	1	0	0	1
241	$F1	1	1	1	1	0	0	0	1
40	$28	0	0	1	0	1	0	0	0
188	$BC	1	0	1	1	1	1	0	0
70	$46	0	1	0	0	0	1	1	0
131	$83	1	0	0	0	0	0	1	1

Такой узор для штриховки можно задать с помощью следующего фрагмента:

В последней главе приводится программа—пример, генерирующая случайные узоры, которые можно запомнить в файле, если они вам понравятся.

GetFillPattern(Var fillPattern: FillPatternType);

Выдает шаблон штриховки, установленный последним вызовом процедуры *SetFillPattern.*

GetFillSettings(Var FillInfo: FillSettingsType);

Выдает текущий шаблон и цвет, установленные процедурами *SetFillStyle* и *SetFillPattern.*

```
Type
    FillSettingsType = Record
                         pattern: Word,
                         colors: Word
                       End;
```

FloodFill(x, y: Integer; border: Word);

Штрихует область вокруг точки с координатами *(x, y)* до границы цвета *border*, используя текущий шаблон и цвет.

FillEllipse(x, y: Integer; xRadius, yRadius: Word);

Чертит и штрихует эллипс текущим цветом и шаблоном.

FillPoly(numPoints: Word; Var polyPoints);

Чертит и штрихует многоугольник с *(numPoints-1)* количеством углов, координаты которых задаются массивом типа *PointType*.

Sector(x, y: Integer; stAngle,endAngle, xRadius, yRadius: Word);

Чертит и штрихует сектор эллипса с центром в точке *(x,y)* от начального угла *stAngle* до конечного *endAngle*, используя *xRadius* и *yRadius* как горизонтальную и вертикальную оси, соответственно.

PieSlice(x, y: Integer; stAngle, endAngle, radius: Word);

Чертит и штрихует сектор круга с центром в точке *(x,y)* и с радиусом *radius* от начального угла *stAngle* до конечного *endAngle* (цвет и шаблон — текущие).

Bar(xl, yl, x2, y2: Integer);

Чертит и штрихует прямоугольник текущим цветом и шаблоном.

Bar3D(xl, yl, x2, y2: Integer; depth: Word; top: Boolean);

Чертит трехмерный параллелепипед текущим цветом и штрихует переднюю грань текущим шаблоном. Параметр *top* определяет способ вывода верхней грани: если *top=True*, грань вырисовывается, иначе — нет. *depth* определяет глубину третьего измерения.

Организация вывода текста в графическом режиме

OutText(textString:String);

Выводит строку *textString* с текущей позиции курсора.

OutTextXY(x,y: Integer; textString:String);

Выводит строку *textString* с позиции *(x,y)*.

SetTextStyle(font, direction: Word; charSize: Word);

Устанавливает текущий шрифт *(font)*, направление *(direction)* и размер *(charSize)* выводимого текста, используемое в процедурах *OutText* и *OutTextXY*.

Для *шрифтов* определены константы:

```
Const
  DefaultFont=0;                          { Стандартный }
  TriplexFont=1;     { Шрифт триплекс из файла Trip.chr }
  SmallFont=2;
    { Шрифт с маленькими символами из файла Litt.chr }
  SansSerifFont=3;    { Прямой шрифт из файла Sans.chr }
  GothicFont=4;     Готический шрифт из файла Goth.chr}
```

Для *направления* текста определены константы:

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

Размер для каждого шрифта может меняться от 1 до 10 (для стандартного — от 1 до 32). При увеличении размера шрифта теряется четкость изображения, особенно для стандартного шрифта.

SetTextJustify(horiz, vert: Word);

Устанавливает горизонтальное *(horiz)* и вертикальное *(vert)* выравнивание текста относительно позиции курсора, используемое в процедурах *OutText* и *OutTextXY*. В качестве параметров можно использовать константы:

```
Const
LeftText=0;       { Вывод справа от курсора }
CenterText=1;     { Курсор в центре вывода }
```

```
RightText=2;      { Вывод слева от курсора }
BottomText=0;     { Вывод сверху от курсора }
TopText=2;        { Вывод снизу от курсора }
```

TextHeight(textString: String): Word;

Возвращает высоту строки *textString* в пикселях.

TextWidth(TextString: string): Word;

Возвращает ширину строки textString в пикселях.

GetTextSettings(Var textInfo: TextSettingsType);

Выводит текущий шрифт, направление, размер и выравнивание текста в структурированной переменной *textInfo*, установленные процедурами *SetTextStyle* и *SetTextJustify*.

```
Type
   TextSettingsType = Record
{шрифт}                          font:Word;
{направление}                    direction:Word;
{размер}                         charSize:Word;
{горизонтальное выравнивание}    horiz: Word;
{вертикальное выравнивание}      vert:Word
                                 End;
```

Управление графическими изображениями.

ImageSize(xl, y1, x2, y2: Integer): Word;

Возвращает число байтов, требуемое для сохранения прямоугольной области экрана.

GetImage(xl, yl, x2, y2: Integer; Var map);

Сохраняет образ изображения в рамке прямоугольной области в буфер оперативной памяти *map*.

PutImage(x, y: Integer; Var map; method: Word);

Выводит образ изображения из буфера *map* на экран в точке с координатами *(x, y)* способом накладывания *method*. Для *способа*

объявлены константы, которые определяют метод, по которому будет выводиться новое изображение относительно старого, уже имеющегося на экране, с использованием логического принципа:

```
Const
    NormalPut=0;    {Заменяет старое изображение новым.}
    XorPut=1;  {Принцип логического исключающее ИЛИ.}
    OrPut=2;                {Принцип логического ИЛИ.}
    AndPut=3;               {Принцип логического ДА.}
    NotPut=4; {Принцип логического НЕТ (т.е. инверсия) .}
```

9.5. Формирование модулей (Units) пользователя.

Возможность модульного программирования, которое позволяет формировать самостоятельные подпрограммы, решающие частные задачи, является мощным механизмом. Мы уже рассмотрели процедуры и функции стандартных модулей. Для этого в самом начале главной программы нужно было указать имя соответствующего модуля и все подпрограммы, а также переменные и константы, определенные внутри того модуля, становились доступными. Обращение к модулю осуществляется с помощью строки:

```
Uses Имя-модуля;
```

Например,

```
Uses Crt, Graph;
```

Турбо-Паскаль также допускает использование процедур и функций, определенных в пределах одной программы, другими пользователями. Эта возможность обеспечивается механизмом *модулей* (Unit) и заключается в следующем: программист формирует процедуры и функции в пределах какой-либо задачи, оформляет их как самостоятельные блоки и проводит полную тестировку, затем тексты подпрограмм можно уже сохранить в отдельном файле, следуя правилам формирования модулей. Схематически структуру модуля можно представить в виде:

```
Unit имя-модуля;
```

```
Interface
{ Блок описаний }
Implementation
{ Блок реализации }
[
Begin
{ Исполняемая часть}
]
End.
```

Формирование модуля начинается с заголовка, который содержит ключевое слово *Unit* и имя модуля, по которому в дальнейшем будет осуществляться обращение к нему и которое должно совпадать с физическим именем файла, где он будет храниться, т.е. имя файла будет «имя-модуля.pas». По этому имени данный модуль может вызываться из других модулей или главной программы с помощью строки:

Uses Имя-модуля;

Затем идет часть *Interface.* Здесь объявляются все типы, переменные, константы, а также процедуры и функции (только заголовки), которые будут использоваться и станут доступными в главной программе и в других модулях.

Часть *Implementation* содержит тела всех процедур и функций, заголовки которых были перечислены в блоке *Interface* со строгим сохранением порядка. Причем, в блоке реализации в заголовках подпрограмм могут опускаться список формальных параметров и тип для функций, так как они уже были описаны в части *Interface.* Если заголовок всё-таки приводится полностью, то он должен точно совпадать с заголовком в блоке описаний.

Модуль завершается необязательной выполняемой частью. Если она отсутствует (вместе со словом *«Begin),* тогда модуль завершается ключевым словом *«End.» с* точкой.

Обычно выполняемая часть предназначена для предварительной подготовки среды главной программы. Она может служить, например, для инициации графического режима, открытия файлов и

их подготовки для чтения/записи, осуществления связи с различными техническими средствами или другими компьютерами из сети и др.

После того, как текст модуля сохранен в файле «имя-модуля.pas», его надо откомпилировать. В Турбо-Паскале определены три режима для компиляции модулей:

Compile
Make
Build

Они различаются только способом связи главной программы и компилируемого модуля,

В режиме «Compile» все модули, перечисленные в главной программе в строке «Uses...», должны быть предварительно откомпилированы и сохранены в файле «имя-модуля.tpu» (расширение является аббревиатурой от «Turbo Pascal Unit»). Файл с расширением *«tpu»* создается в результате компиляции файла «имя-модуля.pas».

В режиме «Make» компилятор проверяет наличие всех *tpu*-файлов, объявленных в главной программе. Если какой-либо из них не обнаружен, система пытается отыскать исходный текст модуля и если «имя-модуля.pas» файл найден, приступает к его компиляции. Кроме этого, в режиме «Make» система следит за возможными изменениями, внесенными в исходный вариант модуля. Если различия обнаружены, то независимо от того, существует ли на диске конечный файл «имя-модуля.tpu», система предварительно компилирует исходный текст модуля до компиляции главной программы.

В режиме «Build» все существующие tpu-файлы игнорируются, система ищет только исходные тексты модулей в «*.pas» файлах и компилирует их. В этом случае при компиляции автоматически учитываются все изменения, сделанные в модулях.

Рассмотрим пример модуля, который инициирует графический режим, вычисляет некоторые характеристики данного монитора и выводит информацию на экран.

```
Unit init_gr;

{**************************}
Interface
{**************************}

Uses Crt, Graph;
Var half_maxX, half_maxY, dr,rej,error: Integer;

Function half(x: Integer): Integer;
{**************************}
Implementation
{**************************}

Function half(x:Integer): Integer;
Begin Half:=x Div 2; End;
{**************************************}
{**  Начало выполняемой части  **}
{**************************************}
Var sl,s2:String;
BEGIN
    dr:=Detect;
    { Detect=0 — константа из модуля «Graph» }
    InitGraph(dr,rej,'C:\TP');
    error:=GraphResult;
    If error <>0
    Then
    Begin
      Writeln(' Ошибка инициации!');
      Writeln(GraphErrorMsg(error));
      Halt
    End
    Else
    Begin
      Str(GetMaxX,sl); Str(GetMaxY,s2);
      OutText(' Разрешение экрана — ' + s1 +' x' + s2 + ', ');
      half_maxX:=half(GetMaxX);
```

```
        half_maxY:=half(GetMaxY);
        Str(half_maxX,sl); Str(half_maxY,s2);
        OutText('Координаты центра — (' + s1 +',' + s2 +').');
        SetColor(5);
        OutTextXY(200,100,'It is *** GRAPH *** mode !!!');
    End
END.
```

Этот исходный код должен храниться в файле под именем «init_gr.pas». В результате компилирования генерируется файл «init_gr.tpu». Для обращения к модулю в тексте главной программы нужно указать:

```
Uses Init_Gr;
```

Каким образом будет выполняться, например, следующая программа:

```
Uses Crt, Graph, Init_Gr;
Var c:Char;
Begin
    c:=ReadKey;   { Задержка для просмотра экрана }
    { Тело главной программы }
    CloseGraph;
End.
```

Сначала, из-за модуля *init_gr*, инициируется графический режим и выводится на экран соответствующая информация, затем начинает работать главная программа, которая завершается оператором «CloseGraph;», закрывающим графический режим.