Морозов А. А.

Программирование на Turbo Pascal 7.0 (Часть 2)

Предисловие

Настоящее учебное пособие содержит описание входного языка системы программирования Turbo Pascal 7.0 Этот язык по своей идеологии отвечает современной методике и технологии изготовления программ любой сложности.

Алгоритмический язык предназначен для точного и полного описания вычислительных алгоритмов и используемых ими структур данных. С помощью *транслятора* такие описания – *программы* – могут быть автоматически преобразованы в эквивалентные программы на цифровом языке конкретной вычислительной машины.

Программа на алгоротмическом языке – это множество *предложений* некоторого *алфавита* (набора основных символов). Предложения и их части записываются в соответствии с правилами, составляющими *синтаксис* языка. Смысл синтаксически правильных предложений определяет *семантика* языка.

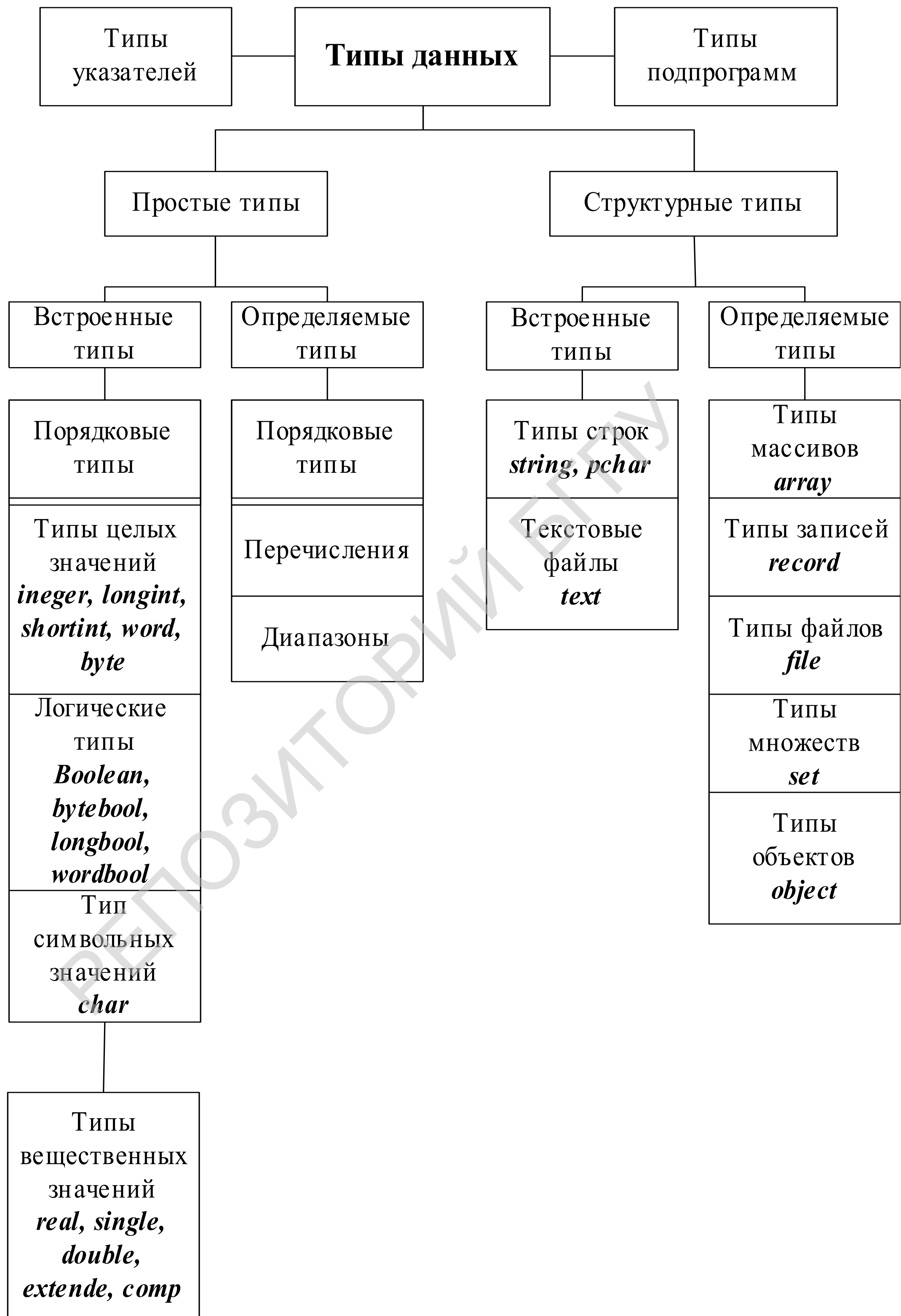
В Турбо Паскале предложения представляют собой последовательности *слов* – идентификаторов, чисел, литерных строк, операций и разделителей. Из предложений строятся все конструкции языка, такие, как операторы или описания констант, переменных, типов данных, подпрограмм и модулей.

Во второй части пособия представлены структурные типы данных языка Турбо Паскаль (кроме типов объектов), типы указателей, подпрограммы и модули. Рассмотрены следующие встроенные (реализованные вместе с транслятором) процедуры и функции из стандартных модулей *System* и *Crt:*

Addr
Append
Assign
Close
Dispose
Eof
Eoln
Erase
Exclude
Exit
FilePos
FileSize
FillChar
FreeMem
GetMem
Halt
Include
IOResult
KeyPressed
Move
New
Read
ReadKey
Readln
Rename
Reset
Rewrite
Seek
SiseOf
Truncate
Write
Writeln

РЕПОЗИТОРИЙ БГПУ

Классификация типов в языке Turbo Pascal 7.0



РЕПОЗИТОРИЙ БГПУ

Типы массивов

Тип массива определяет множество последовательностей, называемых *массивами*, каждая из которых состоит из фиксированного числа компонентов некоторого типа.

Тип массива изображается с помощью служебных слов ***array of*** и записывается в виде:

array *[T1]* ***of array*** *[T2]* ***of*** *...* ***array*** *[Tn]* ***of*** *T*

Здесь *T1, T2 ... Tn* – типы *индексов* массива – идентификаторы или изображения любых порядковых типов, кроме ***integer, word, longint***.

T – тип компонентов или *базовый тип* массива – идентификатор типа или изображение типа.

Число типов индекса *n* называется *размерностью* массива. Различают *одномерные* и *многомерные* (*двухмерные, трехмерные* и т.д.) массивы. На практике массивы размерностью более трех используются редко.

Порядковый тип индекса определяет число значений этого индекса. Число компонентов массива равно произведению числа значений индекса по каждому измерению и поэтому известно во время трансляции программы.

П р и м е р ы описаний типов массивов:

type
Polynom = ***array*** *[0..9]* ***of real;***
A = ***array [byte] of*** *0..1;*
Color = (Red, Green, Blue); B = ***array*** *[Color]* ***of Boolean;***
C = ***array*** *[-1..1]* ***of array [byte] of char;***

Тип *Polynom* определяет одномерные числовые массивы, состоящие из десяти вещественных значений со значениями индекса *0,1 ... 9* (эти массивы могут представлять коэффициенты полиномов степени 9). Тип *A* определяет одномерные числовые массивы, состоящие из *256* нулей и единиц со значениями индекса *0,1 ... 255*. Тип *B* определяет одномерные логические массивы, состоящие из трех логических значений со значениями индекса *Red, Green, Blue*. Тип *C* определяет двухмерные символьные массивы, состоящие из *3*256 = 768* символьных значений со значениями индексов *–1,0,1* и *0,1 ... 255*.

Одномерные массивы с целыми индексами называются *векторами.* Двухмерные массивы с целыми индексами называются *матрицами.*

Тип ***array*** *[T1]* ***of array*** *[T2] ...* ***of array*** *[Tn]* ***of*** *T* имеет сокращенную форму записи ***array*** *[T1,T2,...,Tn]* ***of*** *T*

РЕПОЗИТОРИЙ БГПУ

Например, для двухмерного массива возможны следующие эквивалентные способы изображения типа массива:

array *[T1]* ***of array*** *[T2]* ***of*** *T*

array *[T1,T2]* ***of*** *T*

В случае трехмерного массива приемлемы способы изображения типа массива:

array *[T1]* ***of array*** *[T2]* ***of array*** *[T3]* ***of*** *T*

array *[T1]* ***of array*** *[T2,T3]* ***of*** *T*

array *[T1,T2]* ***of array*** *[T3]* ***of*** *T*

array *[T1,T2,T3]* ***of*** *T*

П р и м е р :

type *A* = ***array*** *[1..5]* ***of array*** *[1..4]* ***of real;***

Тип *A* (массив массивов) определяет векторы, состоящие из 5 компонентов, каждый из которых, в свою очередь, состоит из 4 числовых значений. Эти векторы можно рассматривать как матрицы из 5 строк и 4 столбцов:

type *A* = ***array*** *[1..5,1..4]* ***of real;***

Индексированные переменные

Отдельные компоненты массива (*элементы массива*) в программе обозначаются с помощью *переменной с индексами* или *индексированной переменной.*

Пусть *a* – массив, тип которого изображается как

array *[T1]* ***of array*** *[T2] ...* ***of array*** *[Tn]* ***of*** *T*.

Тогда *a[i1][i2]...[in]* – переменная с индексами.

Здесь *ik* – *индексное выражение*, тип которого должен быть совместим с порядковым типом индекса *Tk.* Тип индекса определяет допустимый диапазон его значений.

П р и м е р ы :

```
type
  Color = (Red, Green, Blue);
  Matrix = array [0..4] of array [0..4] of Boolean;
var
  a : array [1..10] of integer;
  b : array [char] of Color;
  c : array [Color] of Color;
  d : Matrix;
```


РЕПОЗИТОРИЙ БГПУ

В пределах действия этих описаний возможны присваивания:

a[1]:= 1; a[2]:= a[1]; b['A']:= Red; c[Blue]:= Blue; d[1][1]:= TRUE

П р и м е р ы *сложных индексов:*

a[i+1] a[a[1+f(x)]] b[Chr(0)] c[Pred(Blue)] d[i+1][j-1]

Индексированная переменная *a[i1][i2]...[in]* имеет сокращенную форму записи *a[i1,i2 ... in]*.

Например, для двухмерного массива возможны следующие эквивалентные способы записи компонентов массива:

a[i1][i2]
a[i1,i2]

В случае трехмерного массива приемлемы способы записи компонентов массива:

a[i1][i2][i3]
a[i1][i2,i3]
a[i1,i2][i3]
a[i1,i2,i3]

Замечания. В Турбо Паскале в отношении индексированных переменных принят порядок вычисления индексных выражений слева направо. Например, для матриц сначала вычисляется первый индекс – номер строки, а затем второй – номер столбца.

Значения индексов должны находиться в пределах, определяемых их типом в описании массива. Если оттранслировать программу с ключом *{$R+}*, то транслятор вставит команды для проверки индексов. Этот ключ обычно используется на этапе отладки программы.

Операции над массивами. Если *a* и *b* – массивы, типы которых эквивалентны, то возможно присваивание *a:= b*. В Турбо Паскале это единственное возможное действие над массивами в целом.

Ограничения. Любой массив в Турбо Паскале занимает непрерывный блок памяти, размер которого не может превышать 64К байтов. Этот предел может быть достигнут либо из-за слишком большого числа компонентов массива, либо из-за большого размера этих компонентов. Отсюда – ограничение на порядковый тип индексов массива.

Символьные векторы. Тип ***array*** *[1..n]* ***of char***, *n > 1* определяет одномерные символьные массивы с целыми индексами – *символьные векторы.* Их можно рассматривать как значения типа ***string****[n]*, представляющие строки постоянной длины *n*. Поэтому символьные векторы и их элементы могут использоваться в строковых выражениях там, где могут использоваться строковые пе-

РЕПОЗИТОРИЙ БГПУ

ременные и константы. Операция сцепления строк + для символьных векторов не допускается.

П р и м е р ы :

var *s :* ***string****;* *a :* ***array*** *[1..5]* ***of char****;*

В пределах действия этих описаний возможны присваивания:

a:= ‘12345’; s:= a; s:= a[2]

Присваивания *a:= ‘1234’* ; *a:= s; a:= s[2]* не допускаются.

Типы множеств

Тип множеств определяет множество всех подмножеств (включая пустое) из значений некоторого порядкового типа, называемого *базовым типом*. Тип множеств изображается с помощью служебных слов ***set of*** и записывается в виде:

set of *тип*

Здесь *тип* – идентификатор или изображение базового типа. Базовым типом множеств могут быть любые порядковые типы, содержащие не более 256 значений с порядковыми номерами в интервале от 0 до 255, их диапазоны, а также перечисления, содержащие не более 256 значений.

П р и м е р ы :

```
type
  T = set of 1..3;
  DayOfWeek = set of (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
var
  M : DayOfWeek; Letters : set of char;
```

Значениями типа *T* являются множества *{1,2,3}*, *{1,2}*, *{1,3}*, *{2,3}*, *{1}*, *{2}*, *{3}*, *{ }*. Базовым типом для *T* является диапазон *1..3*. Значениями типа *DayOfWeek* являются множества, составленные из идентификаторов перечисления – базового типа этих множеств. Базовым типом множества *Letters* является тип ***char***.

Если базовый тип принимает *n* различных значений, то тип множеств для него будет иметь 2^n значений (различных множеств).

Множества в программе записываются в виде *констант* и *конструкторов множеств.*

Множества-константы состоят из списка констант и диапазонов базового типа внутри квадратных скобок. Пустой список соответствует пустому множеству. *Конструкторы множеств* содержат внутри квадратных скобок не только константы базового типа, но и произвольные выражения этого типа. Любое множество можно изобразить многими эквивалентными способами.

РЕПОЗИТОРИЙ БГПУ

П р и м е р ы правильной записи множеств:

[ ] – пустое множество

*[**false, true**]*

[2..1] – это множество эквивалентно множеству [ ]

[1, 3..5] [1, 3..2+3] [3, 4, 5, 1, 1] – эти множества эквивалентны

['a', 'b', Chr(99)]   ['a', #98, 'c',] – эти множества эквивалентны

Для вычисления множеств служат *выражения,* которые состоят из множеств-переменных, множеств-констант, конструкторов множеств и операций над ними.

Операции над множествами. Над множествами одного базового типа определены операции:

+	*объединение множеств*
-	*разность множеств*
*	*пересечение множеств*

Для сравнения множеств используются операции:

=	*равно*	<=	*содержится в*
<>	*не равно*	>=	*содержит*

Для проверки принадлежности некоторого значения множеству имеется операция ***in***.

Пусть *M* и *N* – множества.

Множество *M+N* состоит из элементов, принадлежащих хотя бы одному из множеств *M* и *N*.

Множество *M-N* состоит из элементов множества *M*, не являющихся элементами множества *N*.

Множество *M*N* состоит из общих элементов множеств *M* и *N*.

П р и м е р ы :

Множество *[0..1]+ [2]* эквивалентно множеству *[0..2]*

Множество *['a'.. 'z']- ['a']* эквивалентно множеству *['b'.. 'z']*

Множество *[ ]- [2]* эквивалентно множеству *[ ]*

Множество *[0..1]* [2]* эквивалентно множеству *[ ]*

Пусть *M* и *N* – множества, *e* – значение базового типа.

Сравнение *M = N* справедливо, если множества *M* и *N* состоят из одних и тех же элементов.

Сравнение *M <> N* справедливо, если в множествах *M* и *N* есть различные элементы.

Сравнение *M <= N* справедливо, если *M* – пустое множество или если каждый элемент множества *M* принадлежит множеству *N*.

РЕПОЗИТОРИЙ БГПУ

Сравнение *M >= N* справедливо, если *N* – пустое множество или если каждый элемент множества *N* принадлежит множеству *M*.

e ***in*** *M* справедливо, если *e* является элементом множества *M*.

П р и м е р ы :

var *M :* ***set of*** *(Red,Green,Blue);*

Пусть *M = [Red,Green]*

выражение	**результат**	**выражение**	**результат**
Green ***in*** *M*	***true***	*M <> [Red]*	***true***
[ ] <= M	***true***	*M = [ ]*	***false***

Справедливы соотношения:

e ***in*** *(M+N) = (e* ***in*** *M)* ***or*** *(e* ***in*** *N)*

e ***in*** *(M-N) = (e* ***in*** *M)* ***and not****(e* ***in*** *N)*

e ***in*** *(M*N) = (e* ***in*** *M)* ***and*** *(e* ***in*** *N)*

Замечание. В памяти компьютера множества представлены машинными словами из двух байтов: i - й бит слова равен 1, если i принадлежит множеству, и равен 0 в противном случае. Операции над множествами реализованы как логические операции над битами.

Процедуры Include и Exclude. Пусть *M* – множество значений базового типа *T*, *n* – выражение типа *T*.

Процедура ***Include(M,n)*** включает в множество *M* значение *n*.

Процедура ***Exclude(M,n)*** исключает из множества *M* значение *n*.

Include(M,n) эквивалентно *M:= M+[n]*

Exclude(M,n) эквивалентно *M:= M-[n]*

Типы записей

Тип записей определяет множество последовательностей, называемых *записями*, каждая из которых состоит из фиксированного числа компонентов – *полей записи*. Поля, обозначаемые идентификаторами, являются в общем случае переменными различных типов.

Тип записей изображается с помощью служебных слов ***record*** и ***end***:

record
  список полей : тип;
  список полей : тип;
  ...
  список полей : тип
end

Здесь *тип* – идентификатор или изображение типа. Перечисление в *списке полей* – через запятую. Все идентификаторы полей записи должны быть различны.

РЕПОЗИТОРИЙ БГПУ

Записи используются для представления структурных данных, например, таких как “комплексное число”, “точка на плоскости”, “прямая линия” или “дата”.

П р и м е р ы описаний типов записей:

type
Complex = ***record*** *Re* : ***real***; *Im* : ***real end;***

или в сокращенной форме

Complex = ***record*** *Re, Im* : ***real end;***

Записи типа *Complex* – пары вещественных значений, которые можно рассматривать как комплексные числа.

```
type
  Point = record x, y : real end;
  Direct = record a, b : Point end;
  Date = record
           Day : 1..31;
           Month : (jan,feb,mar,apr,maj,june,july,aug,sept,oct,nov,dec);
           Year : word
         end;
  T = record  end;
```

Если записи типа *Point* – пары вещественных значений – рассматривать как координаты точек на плоскости, то записи типа *Direct* представляют прямые линии. Тип *Date* соответствует понятию “дата”. Записи типа *T* не содержат компонентов.

Операции над записями. Отдельные компоненты записи – поля – доступны в программе с помощью составных имен:

идентификатор записи **.** *идентификатор поля*

П р и м е р ы :

```
type
  Point = record x, y : real end; Direct = record a, b : Point end;
var
  P, Q : Point;  L : Direct;
```

В пределах действия этих описаний возможны присваивания:

P.x:= 0; P.y:= 0; Q.x:= 1; Q.y:= 1

L.a.x:= P.x; L.a.y:= P.y; L.b.x:= 1; L.b.y:= 0

В последнем случае *операция выбора поля* **.** (точка) применяется сначала по отношению к компонентам записи *L,* а затем к компонентам записей *a* и *b* (слева направо).

Если *r* и *s* – записи, типы которых *эквивалентны*, то допустимо присваивание *r*:= *s*. Это единственное возможное действие над записями в целом.

РЕПОЗИТОРИЙ БГПУ

Для записей *L* , *P* и *Q* из предыдущего примера возможны присваивания: *L.a:= P; L.b:= Q*

Оператор над записями with

Оператор над записями ***with*** служит для более наглядной и эффективной организации работы с переменными, имеющими структуру записи.

Оператор над записями записывается в виде:

with *список переменных* ***do*** *оператор*

Все переменные в списке должны иметь структуру записи. Перечисление в списке – через запятую.

Пусть *r* – запись, *S* – оператор.

with *r* ***do*** *S* означает выполнение оператора *S*, при этом внутри *S* можно записывать идентификаторы полей записи *r*, опуская предшествующее обозначение самой переменной *r* и точку.

П р и м е р :

var
  complex ***: record*** *re, im* ***: real end;***

Оператор

begin *complex.re:= 1; complex.im:= 1* ***end***

эквивалентен оператору

with *complex* ***do begin*** *re:= 1; im:= 1* ***end***

Оператор ***with*** удобно применять, когда на небольшом отрезке программы многократно используются компоненты некоторой записи. Он также может приводить к повышению эффективности программы, т. к. адрес записи в этом случае вычисляется только один раз.

Оператор

with *r1, r2 ... rn* ***do*** *S*

является сокращенной формой записи оператора

with *r1* ***do with*** *r2* ***do ... with*** *rn* ***do*** *S*

Записи с вариантами

Записи с вариантами позволяют определить несколько вариантов структуры записей, относящихся к одному и тому же типу. Они содержат *фиксированную* и *вариантную части:*

record
  фиксированная часть
  вариантная часть
end

РЕПОЗИТОРИЙ БГПУ

Общая часть по структуре не отличается от обычной записи и может отсутствовать.

Вариантная часть, которая должна располагаться после общей, изображается с помощью служебных слов ***case of*** и записывается в виде:

```
case дискриминант : тип of
   список меток : (список полей);
   список меток : (список полей);
   ...
   список меток : (список полей)
end
```

Дискриминант или *селектор вариантов* – это особый компонент (отдельное поле) записи. *Метки* – константы, тип которых должен быть совместим с типом дискриминанта. Перечисление в списке меток – через запятую.

Дискриминант определяет *действующий (активный)* вариант записи. Действующим будет вариант, список меток которого содержит значение дискриминанта.

Структура *списка полей* внутри круглых скобок аналогична структуре списка полей фиксированной части записи. Этот список может быть пустым.

Любой вариант, в свою очередь, может иметь вариантную часть.

Записи с вариантами удобны для представления информации, частично различающейся по своей структуре. Они упрощают разработку процедур обработки такой информации, а также экономят память.

П р и м е р ы :

```
type
   TFigure =  (Square, Triangle, Circle);
   Figure = record
                 x, y : real;
            case tag : Tfigure of
               Square: (Side : real);
               Triangle: (Side1, Side2, Angle : real);
               Circle: (Radius : real)
            end;
var F : Figure;
```

В пределах действия этих описаний возможны присваивания:

```
F.x:= 1; F.y:= 1; F.tag:= Circle; F.Radius:= 1
if F.tag = Square then F.Side:= 2
```

Заголовок вариантной части

case *дискриминант : тип* ***of***

можно записать также в виде:

case *дискриминант* ***of*** или ***case*** *тип* ***of***

РЕПОЗИТОРИЙ БГПУ

В первом случае на месте дискриминанта используется переменная, которая не является полем определяемой записи. Во втором случае запись не содержит дискриминанта. Все варианты такой записи считаются действующими, а метки никак не используются.

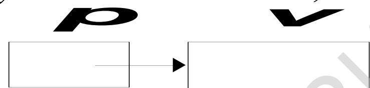
П р и м е р :

```
var mem : record
            case byte of
              0 : (byt : array [0..1] of byte);
              1 : (wrd : word);
            end;
```

Во внутреннем представлении записей с вариантами, принятом в Турбо Паскале, первое поле каждого варианта начинается с одного и того же байта памяти. Запись *mem* имеет два варианта, каждый из которых занимает в памяти один и тот же участок из двух байтов. Поэтому оператор *mem.wrd:= 1* и операторы *mem.byt[0]:= 1; mem.byt[1]:= 0* эквивалентны. Турбо Паскаль не содержит никаких средств автоматического контроля за правильной работой с вариантами записей.

Типы указателей

Указатель или *ссылка* – это переменная, содержащая некоторый адрес в памяти компьютера. Если указатель *p* содержит адрес переменной *v*, то говорят, что *p указывает* на *v* или, что то же, *p ссылается* на *v:*



Тип указателей определяет множество указателей, каждый из которых ссылается на отдельную переменную некоторого *базового* типа. Этому множеству принадлежит также *пустой указатель*, который по определению не ссылается ни на какую переменную. Пустой указатель, общий для всех типов указателей, обозначается стандартным идентификатором ***nil***.

Тип указателей записывается с помощью знака ^ в виде:

^ *тип*

Здесь *тип* – идентификатор базового типа. Изображение типа не допускается. Возможны указатели на значения любых типов.

П р и м е р ы описаний типов указателей:

```
type
  PReal = ^real;
  Buffer = string[80]; PBuffer = ^ Buffer;
  TArray = array[1..10] of PBuffer; PArray = ^TArray;
var
  p : PReal; q : PBuffer; m : PArray; r : ^Boolean;
```


РЕПОЗИТОРИЙ БГПУ

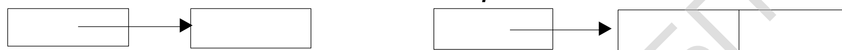
Операции над указателями. Пусть *v* – переменная типа *T*, *p* – указатель типа ^*T*. Адрес *v* можно получить с помощью операции @ (*адрес*) и присваивания *p:= @v* (*p* присвоить адрес *v*). В результате *p* будет ссылаться на переменную *v*. Значение *v* доступно через указатель *p* с помощью операции *косвенной адресации* или *раскрытия ссылки p^* .

П р и м е р ы :

```
type
   vector = array[1..2] of byte;  ref = ^byte;
var
   v : byte;  m : vector;  p, r : ref;  q : ^ vector;
```

В пределах действия этих описаний возможны присваивания:

```
p:= nil;  r:= p;
p:= @v;  p^:= 1;       { эквивалентно v:= 1}
q:= @m; q^[1]:= 1;   { эквивалентно m[1]:= 1 }
```




```
p:= @m[1]; p^:= 1; { эквивалентно m[1]:= 1 }
```

Присваивание *p:= q* недопустимо, т.к. указатели *p* и *q* принадлежат разным типам.

Можно определить указатели, которые ссылаются на указатели и т. д.

П р и м е р ы :

```
type
   ref = ^byte;
var
   v : byte;  p, q : ref;  pp : ^ref;
```

В пределах действия этих описаний возможны присваивания:

```
p:= @v; pp:= @p
q:= pp^;  q^:= 1  { эквивалентно v:= 1 или  pp^^:= 1}
```

Указатели одного и того же типа можно сравнивать с помощью операций = (*равно*) и <> (*не равно*).

Динамические переменные

Переменные, на которые указывают ссылочные значения, могут появляться во время выполнения программы. Такие переменные называются *динамическими* и широко используются во многих приложениях. Память для них рассматривается как массив байтов, который называется *кучей.*

Динамические переменные создаются с помощью стандартной процедуры *New* и уничтожаются с помощью стандартной процедуры *Dispose* или в результате окончания программы. Они не именуются и доступны только через указатели с помощью операции раскрытия ссылки.

РЕПОЗИТОРИЙ БГПУ

Процедуры для работы с динамической памятью *New* и *Dispose*. Пусть *p* – указатель типа ^*T*, где *T* – базовый тип.

Процедура *New(p)* создает новую переменную типа *T*, значение которой не определено, и присваивает адрес этой переменной в куче указателю *p*.

Процедура *Dispose(p)* – обратная к *New(p)* – уничтожает переменную, на которую ссылается *p* и возвращает память в кучу. Значение указателя *p* при этом не изменяется. Если *p* равно ***nil***, то вызов процедуры приводит к ошибке.

П р и м е р :

```
type
   TRecord = record  x, y : byte end;
var
   p : ^ TRecord; q : ^byte;
```

В пределах действия этих описаний возможны присваивания динамическим переменным:

```
New(q);  q^:= 1;  New(p);  p^.x:= 1;  p^.y:= q^
Dispose(q);  q:= nil;  Dispose(p);  p:= nil
```

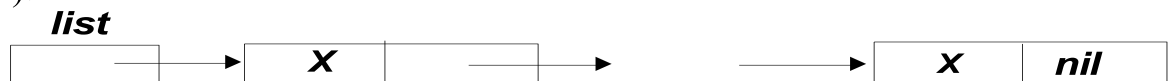
После того как память освобождается, рекомендуется присвоить указателям *q* и *p* значение ***nil***. В противном случае в программе останутся указатели на память в куче, которая может быть перераспределена для других переменных.

С помощью указателей и динамических переменных определяются данные со сложной, изменяющейся структурой.

П р и м е р :

```
type
   ref = ^ Node;
   Node = record  x : real;  next : ref  end;
var
   list, q : ref;
```

Эти описания определяют структуру простого *связанного списка,* который состоит из упорядоченного набора *узлов.* Узлы представлены записями, имеющими два поля. Первое содержит вещественное значение, второе – ссылку на следующий узел в списке. Указатель *list* ссылается на головной узел (начало списка):



В отличие от массивов связанные списки *динамичны* – они могут сжиматься или расти, забирая из кучи столько памяти, сколько необходимо.

Присваивание *list*:= ***nil*** дает пустой список.

Операторы *New(q); q^.next:= list; list:=q* включают новый узел в голову списка.

РЕПОЗИТОРИЙ БГПУ

Операторы *q:= list; list:= q^.next; Dispose(q)* исключают головной узел из непустого списка.

Замечание. Турбо Паскаль разрешает использовать вызов процедуры *New* в форме вызова функции, что в ряде случаев более удобно. В качестве параметра записывается тип используемого указателя.

П р и м е р :

type *ref = ^* ***real****;*
var *p : ref;*
Операторы *New(p)* и *p:= New(ref)* эквивалентны.

Тип указателей pointer

Турбо Паскаль имеет встроенный тип указателей ***pointer****,* который не связан ни с каким базовым типом. Поэтому для указателей этого типа не определена операция косвенной адресации ^ , не применимы процедуры *New* и *Dispose.* В то же время тип ***pointer*** совместим с любым другим типом указателей.

С помощью указателей ***pointer*** можно определить данные, структура которых эквивалентна произвольному ориентированному графу. Указатели этого типа используются также для преобразования других указателей, относящихся к разным типам.

П р и м е р :

type
TArray = ***array****[1..4]* ***of byte****; TRecord =* ***record*** *x, y* ***: integer end****;*
var
q : ^ TArray; r : ^ TRecord; p : ***pointer****;*
В пределах действия этих описаний возможны присваивания:

New(q); q^[4]:= 0; q^[3]:= 1; p:= q; r:= p; r^.y:= 1

Указатель *q,* ссылающийся на массив, преобразуется в указатель *r,* ссылающийся на запись.

Процедуры для работы с динамической памятью *GetMem* и *FreeMem*. Пусть *p* – указатель любого типа, *n* – выражение типа ***word***.

Процедура *GetMem(p, n)* выделяет в куче область памяти размером *n* байтов и присваивает ее адрес указателю *p.*

Процедура *FreeMem(p, n)* – обратная к *GetMem(p, n)* – освобождает занятую область памяти. Значение указателя *p* при этом не изменяется.

Пусть *T* – некоторый базовый тип. Если *p* – указатель типа *^T,* то
New(p) эквивалентно *GetMem(p, SizeOf(T))* или *GetMem(p, SizeOf(p^))*
Dispose(p) эквивалентно *FreetMem(p,SizeOf(T))* или

```
FreetMem(p,SizeOf(p^))
```

Пример:

```
type
   T = array[1..2,1..2] of integer;
var
   p : pointer; m : ^ T;
begin
   GetMem(p, 8);  m:= p;  m^[1,1]:= 1;
end.
```

В этой программе указатель *p* ссылается на блок памяти размером 8 байтов, на который накладывается структура двухмерного массива.

Функция ***Addr(x)*** дает указатель типа ***pointer,*** который ссылается на *x*. Здесь *x* – любая переменная или имя подпрограммы.

Типы файлов

Файлы служат для представления в паскаль-программах *наборов данных,* которые могут храниться на внешних запоминающих устройствах компьютера или отождествляться с потоками данных его устройств ввода и вывода.

Тип файлов определяет множество последовательностей, называемых *файлами*, каждая из которых состоит из компонентов одного типа. Число компонентов называется *длиной файла.* В отличие от массивов это число не постоянно – файлы можно расширять, а также целиком стирать. Файл, не имеющий ни одного компонента, называется *пустым,* его длина равна нулю.

Тип файлов изображается с помощью служебных слов ***file of*** и записывается в виде:

file of *тип*

Здесь *тип* – тип компонентов или *базовый тип* файла – идентификатор или изображение любого типа, кроме типа файлов. Например, могут быть файлы записей, массивов или множеств.

Примеры описаний файлов:

```
type
  IntFile = file of integer;
  CompFile = file of record Re, Im : real  end;
var
   f : IntFile;  g : CompFile; h : file of char;
```

В программе, содержащей эти описания файл *f* будет представлять набор целочисленных данных, файл *g* – набор данных, состоящий из пар вещественных значений.

Файлы значительно упрощают программирование обработки внешних данных, которая сводится в основном к использованию небольшого числа стан-

РЕПОЗИТОРИЙ БГПУ

дартных процедур и функций.

Операции над файлами

Любой файл сначала должен быть *связан* с некоторым набором данных на внешнем устройстве памяти. Затем файл может быть *открыт для записи (формирования)* или *для чтения (просмотра)* этого набора. При необходимости открытый файл можно *закрыть,* а затем снова открыть.

Предполагается, что с каждым открытым файлом связаны *указатель файла,* который определяет его *доступный* компонент, а также специальный маркер – *конец файла*. Началу файла соответствует нулевое значение указателя, концу – значение, равное длине файла.

Новый файл формируется путем добавления компонентов в конец файла. Непустой файл просматривается от начала к концу *(последовательный доступ)* или в произвольном порядке *(прямой доступ)*.

Чтение/запись внешних данных с помощью файлов называется также *вводом/выводом.* По направлению последовательного ввода/вывода различают *входные* и *выходные* файлы.

Пусть *f* – переменная любого типа файлов, *s* – строка.

Процедура *Assign(f, s)* связывает файл *f* с набором данных, определенным строкой *s*. Не допускается, чтобы файл *f* был открыт. Имя набора данных в *s* может включать в себя обозначение дискового устройства и цепочку каталогов.

Процедура *Reset(f)* открывает файл *f,* который должен быть связан с некоторым набором данных. Указатель файла устанавливается в начальное (нулевое) положение.

Процедура *Rewrite(f)* открывает файл *f,* который должен быть связан с новым, формируемым набором данных. Если набор данных уже существовал, то он будет удален и создан заново пустой набор. Указатель файла устанавливается в начальное (нулевое) положение.

Пусть *v* – переменная базового типа для файла *f*.

Процедура *Write(f, v)* записывает в файл *f* значение *v*. При этом файл расширяется, если указатель файла находится в конечном положении. Файл *f* должен быть открыт процедурой *Reset* или *Rewrite*.

Write(f, v1, v2 ... vn) эквивалентно *Write(f, v1); Write(f, v2); ... Write(f, vn)*

Процедура *Read(f, v)* считывает значение из файла *f* в переменную *v*. При этом указатель файла не должен находиться в конечном положении. Файл *f* должен быть открыт процедурой *Reset* или *Rewrite*.

РЕПОЗИТОРИЙ БГПУ

Read(f, v1, v2 ... vn) эквивалентно *Read(f, v1); Read(f, v2); ... Read(f, vn)*

Процедура ***Close(f)*** закрывает файл *f.* В дальнейшем файл *f* может быть связан с каким-либо другим набором данных. Перед завершением программы рекомендуется закрыть все открытые файлы. Нельзя закрыть уже закрытый файл.

Процедура ***Erase(f)*** удаляет набор данных, связанный с файлом *f.* Файл *f* должен быть закрыт.

Процедура ***Rename(f, s)*** заменяет имя связанного с файлом *f* набора данных на имя, определенное строкой *s.* Файл *f* должен быть закрыт. В строке *s* нельзя указывать имя устройства и каталог, а также имя уже существующего набора данных.

Логическая функция ***Eof(f)*** дает значение *TRUE,* если достигнут конец файла *f*, и *FALSE* – в противном случае.

Целочисленная функция ***FileSize(f)*** дает длину (число компонентов) файла *f*. Файл *f* должен быть открыт.

Целочисленная функция ***FilePos(f)*** дает значение указателя файла *f,* которое всегда находится в диапазоне от *0* до *FileSize(f).* Файл *f* должен быть открыт.

Процедура ***Truncate(f)*** удаляет хвостовую часть файла *f,* начиная с позиции, определяемой функцией *FilePos.* Файл *f* должен быть открыт.

Пусть *n* – выражение целого типа.

Процедура ***Seek(f,n)*** устанавливает указатель файла *f* в положение, определяемое значением *n.* Файл *f* должен быть открыт. Значение *n* не должно выходить за пределы диапазона от *0* (начало файла) до *FileSize(f)* (конец файла).

П р и м е р :

```
var
   Data : file of byte;  x : byte;
begin
    Assign(Data, 'A: \ NUMB.DAT');
   Reset(Data);
   while not Eof(Data) do
   begin
      Read(Data, x); Write(x : 2)
   end;
   Close(Data)
end.
```

Эта программа выводит на экран целые значения из файла *Data,* который

РЕПОЗИТОРИЙ БГПУ

связан с находящимся на дисковом устройстве *A:* набором данных *NUMB.DAT*.

П р и м е р :

```
var
   f : file of record x, y : real end;
begin
   Assign(f, 'DATA'); Rewrite(f);
   Writeln(FilePos(f), Eof(f) : 5);
   if  FileSize(f) = 0 then Writeln('Файл пуст');
   Close(f)
end.
```

После выполнения процедуры *Rewrite* файл набора данных *DATA* пуст. Указатель файла находится одновременно в его начале и в конце. Поэтому программа выводит на экран значения *0* и *TRUE,* а также сообщение “Файл пуст”.

Используя функции *Seek, FileSize* и *FilePos* можно записать достаточно сложные алгоритмы работы с файлами прямого доступа.

П р и м е р ы :

Переместить указатель	**Работа с файлом**
Seek(f, 0)	*Для чтения или замены первого компонента файла f*
Seek(f, FileSize(f))	*Для расширения файла f*
Seek(f, FileSize(f)-1)	*Для чтения или замены последнего компонента файла f*
Seek(f, FilePos(f)+1)	*Для перемещения к следующему компоненту файла f*

Замечание. Турбо Паскаль разрешает определять *файлы без типа,* с помощью которых можно получить прямой доступ к любым внешним наборам данных независимо от их внутренней структуры.

Файлы без типа в этой части справочника не рассматриваются.

РЕПОЗИТОРИЙ БГПУ

Текстовые файлы

Текстовые файлы служат для работы с текстовыми данными и состоят из символьных строк переменной длины. Каждая строка имеет специальный маркер – *конец строки,* который состоит из пары управляющих символов *#13* (возврат каретки) и *#10* (перевод строки). Маркер конца файла состоит из управляющего символа *#26.* Символ *#9* (табуляция) в текстовых файлах обычно заменяет один или несколько пробелов.

Текстовые файлы обозначаются стандартным идентификатором ***text***.

П р и м е р ы :

```
type T = text;
var  f1, f2 : text;   f : T;
```

Логические устройства ввода/вывода. В отличие от других файлов текстовые файлы могут быть связаны как с наборами данных на внешних запоминающих устройствах компьютера, так и с его устройствами ввода/вывода различных типов.

В Турбо Паскале устройства ввода/вывода имеют свои символические обозначения, которые называются *логическими устройствами:*

CON – устройство ввода/вывода *консоль.* При записи данных консоль обозначает экран дисплея, при чтении – клавиатуру.

LPT1, LPT2, LPT3 – печатающие устройства вывода.

PRN – синоним для *LPT1*.

COM1, COM2 – любые устройства ввода/вывода, которые можно подключить к *последовательным коммуникационным портам* компьютера.

AUX – синоним для *COM1*.

NUL – *пустое* устройство ввода/вывода. При записи это логическое устройство рассматривается как приемник данных без их реальной пересылки. При чтении данных поступает признак конца файла. Пустое устройство используется при отладке программ.

Операции над текстовыми файлами

Текстовые файлы обрабатываются строго *последовательно* и могут быть открыты либо только для записи (вывода), либо только для чтения (ввода).

Текстовые файлы можно обрабатывать посимвольно и построчно. В этом – их отличие от символьных файлов *(**file of char**),* которые являются просто последовательностями значений символьного типа.

Текстовые файлы состоят из символьных строк, но могут быть использованы для кодирования целых, вещественных и логических значений. В этом случае операции ввода/вывода автоматически преобразуют данные других типов из их внутреннего представления в символьные строки и обратно.

Текстовый файл может быть связан процедурой *Assign* с набором данных на внешнем устройстве памяти или с одним из логических устройств ввода/вывода. Данные обрабатываются после его открытия одной из процедур *Reset, Rewrite* или *Append.* Файл закрывается процедурой *Close.* Размер компонентов текстовых файлов непостоянен, поэтому процедура *Seek* к ним неприменима.

Для ввода/вывода можно также использовать стандартные текстовые файлы *Input, Output,* и *Lst,* которые связаны, соответственно, с клавиатурой, экраном и устройством печати.

РЕПОЗИТОРИЙ БГПУ

Пусть *f* – текстовый файл, *s* – строка.

Процедура *Assign(f, s)* связывает файл *f* с набором данных или логическим устройством ввода/вывода, определенным строкой *s*. Не допускается, чтобы файл *f* был открыт. Имя набора данных в *s* может включать в себя обозначение дискового устройства и цепочку каталогов.

Процедура *Append(f)* открывает файл *f* для его *расширения.* Файл *f* должен быть связан с некоторым набором данных. Указатель файла устанавливается в конечное положение.

Процедура *Reset(f)* открывает файл *f* для чтения (просмотра). Файл *f* должен быть связан с некоторым набором данных или устройством ввода. Указатель файла устанавливается в начальное положение.

Процедура *Rewrite(f)* открывает файл *f* для записи (формирования). Файл *f* должен быть связан с новым, формируемым набором данных или устройством вывода. Если набор данных уже существовал, то он будет удален и создан заново пустой набор. Указатель файла устанавливается в начальное положение.

Пусть *e* – выражение, тип которого может быть типом целых, вещественных, логических или символьных значений, или типом строк.

Процедура *Write(f, e)* записывает в файл *f* значение *v*. При этом файл расширяется, если указатель файла находится в конечном положении. Файл *f* должен быть открыт процедурой *Rewrite* или *Append.*

Процедура *Write(f, e1, e2 ... en)* записывает в файл *f* несколько значений. Тип каждого из выражений *e1, e2 ... en* независимо от других может быть типом целых, вещественных, логических или символьных значений, или типом строк.

Write(f, e1, e2 ... en) эквивалентно *Write(f, e1); Write(f, e2); ... Write(f, en)*

Процедура *Writeln(f)* записывает в файл *f* маркер конца строки.

Writeln(f) эквивалентно *Write(f, #13, #10)*

Writeln(f, e1, e2 ... en) эквивалентно *Write(f, e1, e2 ... en); Writeln(f)*

Пусть *v* – переменная, тип которой может быть типом целых, вещественных или символьных значений, или типом строк.

Процедура *Read(f, v)* считывает значение из файла *f* в переменную *v*. При этом указатель файла не должен находиться в конечном положении. Файл *f* должен быть открыт процедурой *Reset.*

РЕПОЗИТОРИЙ БГПУ

Процедура *Read(f, v1, v2 ... vn)* считывает из файла *f* несколько значений. Тип каждой из переменных *v1, v2 ... vn* независимо от других может быть типом целых, вещественных или символьных значений, или типом строк.

Read(f, v1, v2 ... vn) эквивалентно *Read(f, v1); Read(f, v2); ... Read(f, vn)*

Процедура *Readln(f)* перемещает указатель файла *f* к началу следующей строки.

Readln(f, v1, v2 ... vn) эквивалентно *Read(f, v1, v2 ... vn); Readln(f)*

Процедура *Close(f)* закрывает файл *f.* Перед завершением программы рекомендуется закрыть все открытые файлы. Нельзя закрыть уже закрытый файл.

Логическая функция *Eof(f)* дает значение *TRUE,* если достигнут конец файла *f*, и *FALSE* – в противном случае.

Логическая функция *Eoln(f)* дает значение *TRUE,* если достигнут конец строки файла *f*, и *FALSE* – в противном случае.

П р и м е р :

```
var Consol : text;
begin
   Assign(Consol, 'CON'); Rewrite(Consol);
   Write(Consol, 5, #13, #10, 1=1, #26, 2);
   Close(Consol)
end.
```

Оператор *Write* в этой программе выводит на экран значения *5* и *TRUE.* Вывод символов *#13* и *#10* означает переход курсора в начало новой строки экрана, символ *#26* (конец файла) завершает вывод.

П р и м е р :

```
var
   Consol : text;  x : byte; r : real; s : string;
begin
   Assign(Consol, 'CON'); Reset(Consol);
   Read(Consol, x, r, s);
   …
   Close(Consol)
end.
```

Оператор *Read* в этой программе считывает данные с клавиатуры. При наборе они разделяются пробелами или нажатием на клавиши <*Tab*> или <*Enter*>. Клавиша <*Enter*> посылает маркер конца строки. При необходимости можно ввести маркер конца файла, который дает комбинация <*Ctrl*><*Z*>. Данные можно набрать на клавиатуре, например, в одной строке экрана в виде *10 -3.14 abc*. После нажатия на клавишу <*Enter*> они автоматически преобразуют-

РЕПОЗИТОРИЙ БГПУ

ся в соответствующие значения, которые присваиваются переменным *x*, *r* и *s*.

Следующая программа подсчитывает число печатных символов *n* и число строк *m* в текстовом наборе данных *T1.TXT*, который читается посимвольно.

```
var
   f : text;  x : char;  n, m : word;
begin
   Assign(f, 'T1.TXT');
   Reset(f);
   n:= 0; m:= 0;
   while not Eof(f) do
   begin
      Read(f, x);
      if not (x  in [#13, #10, #9, #32]) then n:= n + 1;
      if  x = #10 then m:= m + 1;
   end;
   Close(f);
   Writeln(n)
end.
```

Следующая программа подсчитывает число строк *n* в текстовом наборе данных *T1.TXT*, который читается построчно.

```
var
   f : text;  x : char;  n : word;
begin
   Assign(f, 'T1.TXT');
   Reset(f);
   n:= 0;
   while not Eof(f) do
   begin
      Readln(f); n:= n + 1;
   end;
   Close(f);
   Writeln(n)
end.
```

РЕПОЗИТОРИЙ БГПУ

Контроль операций ввода/вывода

При работе с файлами автоматически проверяются ошибки ввода/вывода. Если такая ошибка будет обнаружена, то выполнение программы прекратится и на экране появится сообщение с указанием типа ошибки.

В Турбо Паскале можно отключить автоматический контроль ошибок ввода/вывода, если оттранслировать соответствующие операции с ключом *{$I-*

}. В этом случае программа продолжает выполняться, а проверить результат ввода/вывода можно с помощью стандартной функции *IOResult*.

Функция *IOResult* дает значение *0,* если операция ввода/вывода завершается успешно. В противном случае функция дает ненулевое целое положительное значение, определяющее тип ошибки.

П р и м е р :

```
var
   f : text; Error : word;
begin
   Assign(f, ‘TXT’);
   {$I-} Erase(f); {$I+}
   Error:= IOResult;
   if Error <> 0  then
       Writeln('Набор данных TXT не найден')
   else
   ...
end.
```

С помощью этой программы будет удален набор данных *TXT*. В случае, если указан не существующий набор, на экране появится текст “Набор данных TXT не найден“.

Ошибки операций ввода/вывода:

IOResult	Сообщение	Операция	Пояснение
100	*Disk read error*	*Read*	*Попытка прочитать из файла, который уже полностью прочитан*
101	*Disk write error*	*Write, Writeln, Close*	*Диск, на котором находится набор данных, полностью заполнен*
102	*File not assigned*	*Reset, Rewrite, Append,* *Erase, Rename*	*Файл не связан с набором данных*
103	*File not open*	*Close, Seek, Read, Write,Eof, FilePos, FileSize*	*Файл не открыт*
104	*File not open for input*	*Read, Readln,* *Eof, Eoln*	*Текстовый файл не открыт для ввода*
105	*File not open for output*	*Write, Writeln*	*Текстовый файл не открыт для вывода*
106	*Invalid numeric format*	*Read, Readln*	*Неправильное представление числа*

РЕПОЗИТОРИЙ БГПУ

			в текстовом файле

Стандартные файлы Input, Output и Lst

Для ввода с клавиатуры и вывода на экран в Турбо Паскале обычно используются стандартные текстовые файлы *Input* и *Output.* Эти файлы считаются всегда открытыми, поэтому процедуры *Assign, Reset, Rewrite* и *Close* для них необязательны.

Поскольку стандартный ввод и вывод входит почти в каждую законченную программу, операторы ввода/вывода с файлами *Input* и *Output* имеют сокращенную форму записи:

Полная запись	**Сокращенная запись**
Read(Input, v)	*Read(v)*
Write(Output, v)	*Write(v)*
Readln(Input)	*Readln*
Writeln(Output)	*Writeln*
Eof(Input)	*Eof*
Eoln(Input)	*Eoln*

Для облегчения вывода на печать можно использовать текстовый файл *Lst,* связанный с устройством *LPT1*. Файл *Lst* определен в модуле *Printer.*

П р и м е р :

```
uses Printer;
begin
   Writeln(Lst, ‘Вывод на принтер’);
end.
```

Эта программа напечатает фразу “ Вывод на принтер ”.

РЕПОЗИТОРИЙ БГПУ

Подпрограммы

В ходе решения почти каждой практической задачи можно выделить отдельные этапы или подзадачи. *Подпрограмма* – это часть программы, которая реализует одну из таких частных задач. Для представления входных и выходных данных подпрограммы могут иметь *параметры.*

Подпрограммы широко используются в программировании. Они упрощают разработку больших программ, делают их понятнее и короче, снижают вероятность появления в них ошибок.

Подпрограмму можно выполнить в разных местах общей программы. В соответствии с этим различают *описание подпрограммы* и *обращение к выполнению подпрограммы* или *вызов подпрограммы.*

В Турбо Паскале используются два вида подпрограмм – *процедуры* и *функции.* Процедура реализует некоторый вспомогательный алгоритм. Для обращения к выполнению процедуры служит *оператор процедуры.* Функция реализует алгоритм, который по заданным входным параметрам (*аргументам*) вычисляет некоторое значение. Для обращения к выполнению функции служит *указатель функции,* который используется в выражениях.

Описание процедуры

Описание процедуры определяет ее имя и действие. Описание процедуры состоит из *заголовка*, за которым может следовать *директива*, и *тела процедуры.* Заголовок включает служебное слово ***procedure***, *имя* процедуры, необязательный *список формальных параметров* и записывается в виде:

procedure *имя(список формальных параметров)*

Заголовок процедуры без параметров имеет вид:

procedure *имя*

Имя – это идентификатор. *Формальные параметры* – идентификаторы, которые представляют входные и выходные данные процедуры. Все идентификаторы должны быть различны.

Список формальных параметров может иметь несколько *секций,* которые разделяются точкой с запятой. Отдельная секция параметров записывается в виде:

список идентификаторов : тип

Перечисление в списке идентификаторов – через запятую. *Тип* – это идентификатор типа. Изображение определяемого типа не допускается.

Перед любой из секций формальных параметров могут стоять служебные слова ***var*** или ***const***, которые определяют специальные действия транслятора при реализации *подстановки параметров.*

П р и м е р ы правильной записи заголовков процедур:

```
type
   T : array[1..4] of byte;
procedure OutData;
procedure P(const d : char;  var x : byte);
procedure MinMax(a : real;  b : real;  var M : T);
```

или в сокращенной форме

```
procedure MinMax(a,  b : real;  var M : T);
```

РЕПОЗИТОРИЙ БГПУ

П р и м е р ы неправильной записи заголовков процедур:

***procedure** P()*	*– для процедур без параметров скобки не нужны*
***procedure** P(x : **array**[1..4] **of** **byte**)*	*– в списке параметров не допускается изображение типа*
***procedure** P(x : **real**, y : **real**)*	*– секции параметров должны разделяться точкой с запятой*

Тело процедуры на языке Паскаль представляет собой *блок,* который состоит из *разделов описаний* и *раздела операторов*. Все они по своей структуре аналогичны соответствующим разделам блока программы. Если не заданы директивы *FORWARD* или *EXTERNAL*, то тело процедуры записывается непосредственно после ее заголовка. Разделитель – точка с запятой.

П р и м е р ы описаний процедур:

```
procedure P;  begin   end;
```

Это описание определяет процедуру без параметров, которая не содержит никаких алгоритмических действий.

```
procedure Exchange(var x, y : real);
var t : real;
begin
    t:= x;  x:= y;  y:= t
end;
```

Это описание определяет процедуру обмена значений двух вещественных переменных.

Директивы подпрограмм. *Директива* – это один из стандартных идентификаторов *ASSEMBLER, EXTERNAL, FAR, FORWARD, INLINE, INTERRUPT, NEAR.* Подпрограммы с директивами могут иметь описание не на языке Паскаль. Директивы также дают информацию транслятору о размещении подпрограмм.

Описание функции

Описание функции определяет ее имя, тип результата и действия по вычислению функции. Описание функции состоит из *заголовка*, за которым может следовать *директива*, и *тела функции*. Заголовок включает служебное слово ***function***, *имя* функции, необязательный список *формальных параметров (аргументов)*, тип результата и записывается в виде:

***function** имя(список формальных параметров) : тип*

РЕПОЗИТОРИЙ БГПУ

Заголовок функции без параметров имеет вид:

***function** имя : тип*

Тип – идентификатор любого простого типа, типа указателей, а также тип ***string***. Изображение определяемого типа не допускается.

Структура *списка формальных параметров* аналогична структуре списка формальных параметров процедуры.

В описаниях функций и процедур используются одни и те же *директивы.*

Структура *тела функции* на языке Паскаль в целом аналогична структуре тела процедуры. Необходимо также, чтобы в разделе операторов содержался хотя бы один выполняемый оператор присваивания, в левой части которого записывается имя функции, а справа – соответствующее типу результата выражение. Значение этого выражения дает значение функции. Если ни одно из таких присваиваний не выполняется, то результат функции считается неопределенным.

П р и м е р ы описаний функций:

***function** F **: byte**; **begin** F:= 0 **end**;*

Это описание определяет функцию без параметров, которая дает нулевое значение и содержит только один обязательный оператор присваивания, в левой части которого записывается имя функции.

```
function Maximum(x, y : real) : real;
begin
   if  x > y  then Maximum:= x else Maximum:= y
end;
```

Это описание определяет функцию, которая дает максимальное значение своих двух аргументов.

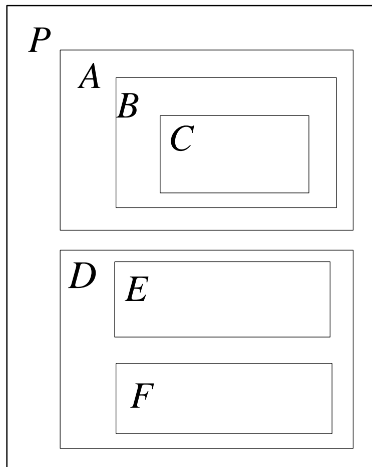
РЕПОЗИТОРИЙ БГПУ

Блочная структура программ

Каждая подпрограмма (процедура или функция), состоящая из заголовка и *блока*, по своей структуре подобна программе. Одна подпрограмма может быть описана внутри другой подпрограммы, поэтому блоки могут вкладываться один в другой. Каждому блоку можно приписать некоторый *уровень*. Если самый внешний блок программы относится к уровню 0, то блоки подпрограмм, которые определены внутри блока уровня 0, относятся к уровню 1 и т.д.

Метки, константы, типы, переменные, процедуры, функции и их параметры *локальны* по отношению к своему блоку. Это означает, что все эти объекты доступны только в пределах блока, внутри которого они определены (включая все внутренние блоки).

П р и м е р блочной структуры программы:



Блоки	**имеют уровень**
P	*0*
A, D	*1*
B, E, F	*2*
C	*3*
Объекты, определенные в блоке	**доступны в блоках**
P	*P, A, B, C, D, E, F*
A	*A, B, C*
B	*B, C*
C	*C*
D	*D, E, F*
E	*E*
F	*F*

Каждый блок вводит новую систему обозначений, действующую только внутри этого блока. Это означает, что для меток, констант, типов, переменных, процедур и функций, а также их параметров, используемых только внутри некоторого блока, можно выбирать любые обозначения (идентификаторы), не заботясь о том, встречались ли такие же обозначения во внешних блоках. При этом локальные идентификаторы подпрограмм и их параметры должны различаться.

П р и м е р :

```
var x, y, z : real;
procedure P(y : byte);
var x : char;
begin ... x ... y ... z ... end;
begin
   ... x ... y ... z ...
end.
```

В этих описаниях один и тот же идентификатор *x* обозначает две разные переменные, один и тот же идентификатор *y* обозначает переменную и формальный параметр. Внутри процедуры *P* идентификатору *x* соответствует описание в блоке *P*, идентификатору *y* – описание в списке параметров *P,* нелокальному идентификатору *z* – описание в блоке программы. Внутри раздела операторов идентификаторам *x, y, z* соответствуют их описания в блоке программы.

Типы подпрограмм

В Турбо Паскале можно описать *типы подпрограмм,* которые изобража-

РЕПОЗИТОРИЙ БГПУ

ются в виде безымянных заголовков процедур и функций.

П р и м е р ы :

```
type
  TProc = procedure;
  TFunc1 = function : string;
  TFunc2 = function (x, y : byte) : real;
var
  F1 : TFunc1;  A : array[1..3] of TFunc2;
```

Тип *TProc* определяет множество процедур без параметров, Тип *TFunc1* –множество строковых функций без параметров. Тип *TFunc* определяет множество вещественных функций с двумя целочисленными параметрами. Идентификаторы формальных параметров таких описаний никак не используются.

Подпрограммы как значения можно присваивать переменным. Такие подпрограммы должны быть оттранслированы с директивой *FAR* или с ключом *{$F+}*.

Например, если в программе имеются описания

```
function Divide(x, y : byte) : real; FAR;
begin
  Divide:= x/y
end;
var
  F1, F2 : TFunc;  a, b : byte;
```

то допустимы присваивания *F1:= Divide; F2:= F1.* Идентификаторы *Divide, F1* и *F2* являются синонимами, поэтому вызовы *Divide(a, b), F1(a, b)* и *F2(a, b)* – эквивалентны.

Ограничения. Нельзя присваивать переменным подпрограммы, которые описаны внутри других подпрограмм, или которые содержат директивы *INTERRUPT* и *INLINE.*

Нельзя также присваивать переменным стандартные процедуры и функции. Это последнее ограничение легко обойти. Например, для *sin(x)* можно определить эквивалентную функцию с другим именем:

```
function Sinus(x :real) : real;
begin
  Sinus:= Sin(x)
end
```

Подстановка параметров

Формальным параметрам подпрограммы (процедуры или функции) соответствуют согласованные по числу и типам *фактические параметры* каждо-

РЕПОЗИТОРИЙ БГПУ

го вызова подпрограммы.

В Турбо Паскале возможны различные способы *подстановки* параметров:

– *подстановка значения;*
– *подстановка переменной;*
– *подстановка константы;*
– *подстановка подпрограммы (процедуры или функции)*.

Подстановка значения. В списке формальных параметров секция *параметров-значений* записывается в виде:

список идентификаторов : тип

При вызове подпрограммы такие формальные параметры рассматриваются как локальные переменные, которым присваиваются значения соответствующих фактических параметров. Фактическими параметрами могут быть выражения, в простейшем случае – константы или переменные.

П р и м е р :

```
var a, b, t : real;
function Maximum(x, y : real) : real;
begin
   if  x > y  then Maximum:= x else Maximum:= y
end;
```

Maximum – функция, которая дает максимальное значение своих аргументов. Действие оператора *t:= Maximum(a, b)* эквивалентно оператору

begin *x:= a; y:= b;* **if** *x > y* **then** *t:= x* **else** *t:= y* **end**

Если фактический параметр – переменная, то на месте формального параметра невозможно присваивание этой переменной нового значения. Поэтому параметры-значения служат для представления только входных данных подпрограммы.

Подстановка значения обеспечивает защиту от ошибочного изменения входных данных. С другой стороны, для каждой локальной переменной выделяется память, и, если фактический параметр – структурная переменная (например: запись или массив), то памяти может потребоваться слишком много.

Подстановка переменной. В списке формальных параметров секция *параметров-переменных* записывается в виде:

var *список идентификаторов : тип*

При вызове подпрограммы такие формальные параметры заменяются соответствующими фактическими параметрами, которые могут быть только переменными. На месте формального параметра возможно присваивание факти-

РЕПОЗИТОРИЙ БГПУ

ческому параметру нового значения. Поэтому параметры-переменные служат для представления как входных, так и выходных данных подпрограммы.

П р и м е р :

```
var a, b : real;
procedure Exchange(var x, y : real);
var t : real;
begin
   t:= x;  x:= y;  y:= t
end;
```

Exchange – процедура обмена значений двух переменных. Действие оператора *Exchange (a, b)* и оператора ***begin*** *t:= a; a:= b; b:= a* ***end*** эквивалентны.

Различие подстановки значения и подстановки переменной хорошо видно на таком примере:

```
var a, b : byte;
procedure P(x : byte; var  y : byte);
begin
    x:= x+1;  y:= y+1;  Writeln(x, y:5)
end;
begin
   a:= 0;  b:= 0;  Writeln(a, b:5);  P(a, b);  Writeln(a, b:5)
end.
```

Программа выводит на экран значения:

```
0     0
1     1
0     1
```

Подстановка константы. В списке формальных параметров секция *параметров-констант* записывается в виде:

const *список идентификаторов : тип*

При вызове подпрограммы на месте таких формальных параметров используются адреса соответствующих фактических параметров, которые могут быть только переменными. При этом присваивание фактическому параметру нового значения не допускается. Этот способ подстановки обеспечивает защиту от ошибочного изменения входных данных и в отличие от подстановки значения не требует выделения дополнительной памяти.

Параметры-константы не могут быть фактическими параметрами в вызовах подпрограмм.

Параметры без типа. Турбо Паскаль допускает параметры-переменные и параметры-константы *без типа*. Это означает, что фактическим параметром в этом случае может быть любая переменная. На месте формального параметра

РЕПОЗИТОРИЙ БГПУ

необходимо явное *преобразование* такой переменной к нужному типу. Транслятор не контролирует правильность использования параметров без типа.

П р и м е р :

```
function Maximum(var a;  n : byte) : real;
type T = array[1..maxint div 6] of  real;
var max : real;  i : byte;
begin
   max:= T(a)[1];
   for i:= 2 to n do  if  max < T(a)[i] then max:= T(a)[i];
   Maximum:= max;
end;
```

Это описание определяет универсальную функцию для вычисления максимального значения из набора вещественных значений.

Пусть в программе, использующей функцию *Maximum,* определены переменные:

```
 var
   m1 : array[1..5] of  real;
   m2 : array[1..3,1..3] of  real;
   w : record  x, y : real end;
```

Тогда вызовы *Maximum(m1, 5), Maximum(m2, 9)* и *Maximum(w, 2)* вычислят, соответственно, максимальные значения в одномерном массиве *m1,* двухмерном массиве *m2* и среди полей записи *w*.

Подстановка подпрограммы. В списке формальных параметров секция *параметров-подпрограмм* записывается в виде:

список идентификаторов : тип подпрограмм

Фактическими параметрами могут быть имена процедур и функций, имеющих необходимое число параметров требуемых типов. Такие процедуры и функции должны быть оттранслированы с директивой *FAR* или с ключом *{$F+}*. Нельзя передавать в качестве параметров подпрограммы, которые описаны внутри других подпрограмм, или содержат директивы *INTERRUPT* и *INLINE.*

П р и м е р ы :

```
type
  TFunc = function(x, y : byte) : byte;
function Add(x, y : byte) : byte; FAR;
begin Add:= x + y end;
{$F+}
function Mult(x, y : byte) : byte;
begin Mult:= x * y  end;
```

РЕПОЗИТОРИЙ БГПУ

```
{$F-}
procedure Table(m, n : byte; F : TFunc);
var  i, j : byte;
begin
   for i:= 1 to m do
   begin
      for j:= 1 to n do Write(F(j, i) : 5); Writeln
   end
end;
```

В программе, использующей эти описания, оператор *Table(10, 10, Add)* будет выводить на экран таблицу сложения 10_10, а оператор *Table(10, 10, Mult)* – таблицу умножения 10_10.

Процедуры Halt и Exit

Для унификации с другими языками программирования в Турбо Паскале предусмотрены процедуры *Halt* и *Exit,* которые можно применять в подпрограммах.

Пусть *w* – выражение типа ***word.***

Процедура *Halt(w)* – *выход из программы* – сразу прекращает выполнение программы. При этом операционной системе передается значение *w – код завершения*. Нулевой код всегда соответствует нормальному завершению программы.

Halt – сокращенная форма записи *Halt(0)*

Процедура без параметров *Exit* – *выход из подпрограммы* – сразу прекращает выполнение любой подпрограммы.

В основной программе *Exit* эквивалентно *Halt.*

Рекурсивные алгоритмы и программы

Турбо Паскаль разрешает *рекурсивный* способ описания подпрограмм через самих себя. Во многих случаях рекурсивные алгоритмы имеют наиболее ясную и компактную форму.

Рассмотрим известное из математики определение факториальной функции с помощью соотношений:

$0! = 1$

$n! = n*(n-1)!\ ,\ n > 0$

РЕПОЗИТОРИЙ БГПУ

Здесь значение *n!* определяется через значение *(n-1)!,* которое определяется через значение *(n-2)!* и т.д., до значения *0!,* которое задано явно.

РЕПОЗИТОРИЙ БГПУ

Функция вычисления факториала на Паскале может быть записана, например, в виде:

```
function Factorial(n : word) : word;
begin
  if  n = 0  then Factorial:= 0
             else Factorial:= n*Factorial(n-1)
end
```

В рекурсивном описании всегда встречается вызов (в качестве вспомогательного алгоритма) самого определяемого алгоритма. Тем самым решение некоторой сложной задачи сводится к решению той же, но проще сформулированной задачи.

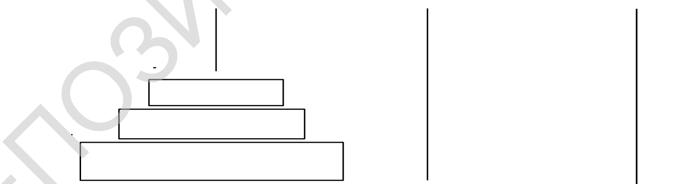
Рассмотрим рекурсивный метод решения на примере *задачи о ханойских башнях:*

Имеется три стержня. На первом из них нанизано *n* дисков различных диаметров, которые образуют пирамиду правильной формы: чем выше расположен диск, тем меньше его диаметр. Требуется переместить всю башню из дисков на второй стержень по следующим правилам:

– можно переносить диски со стержня на стержень только по одному;

– диск нельзя помещать на диск меньшего диаметра;

– для промежуточного хранения дисков можно использовать вспомогательный третий стержень:



РЕПОЗИТОРИЙ БГПУ

Если башня состоит из двух дисков, то решение найти легко, то есть перенести диск:

– с первого на третий (вспомогательный) стержень;

– с первого на второй стержень;

– с третьего на второй стержень.

Рекурсия позволяет легко обобщить это решение на случай произвольного числа дисков. Пусть *Hanoi(n,a,b,c)* обозначает процедуру, которая печатает последовательность всех перемещений *n* дисков. Рассмотрим башню из *n* дисков как пирамиду, состоящую из самого большого диска (основания), на котором находится башня из *n-1* дисков. Тогда рекурсивный алгоритм может быть записан в виде:

```
procedure Hanoi(n,a,b,c : word);
begin
  if n > 0 then
   begin
     Hanoi(n-1,a,c,b); { перенести пирамиду из n-1 дисков с a на c}
     Writeln('перенести диск со стержня ',a,' на стержень',b);
     Hanoi(n-1,c,b,a) { перенести пирамиду из n-1 дисков с c на b}
   end
end
```

При $n = 0$ процедура *Hanoi* не выполняет никаких алгоритмических действий. Если $n > 0$, каждый ее вызов приводит к новым вызовам этой процедуры. После завершения каждого из рекурсивных вызовов *Hanoi(n-1,a,c,b)* печатается сообщение о переносе одного диска. Например, вызов *Hanoi(3,1,2,3)* дает ответ:

перенести диск со стержня 1 на стержень 2
перенести диск со стержня 1 на стержень 3
перенести диск со стержня 2 на стержень 3
перенести диск со стержня 1 на стержень 2
перенести диск со стержня 3 на стержень 1
перенести диск со стержня 3 на стержень 2
перенести диск со стержня 1 на стержень 2

Косвенная рекурсия. В паскаль-программах возможны рекурсивные вызовы процедур или функций, которые заданы с помощью обычных (не рекурсивных) описаний. Примером такой *косвенной* рекурсии является следующая последовательность вызовов:

P1 вызывает P2 вызывает P3 вызывает P1

Возможен и другой случай косвенной рекурсии, когда она возникает в результате подстановки некоторой процедуры или функции на место формального параметра-подпрограммы.

Для того чтобы записать подпрограмму, которая вызывает другую, а та, в свою очередь, рекурсивно вызывает первую, необходимо *опережающее описание* одной из этих подпрограмм.

Опережающее описание подпрограмм. Если в описании подпрограммы задана директива *FORWARD,* то ее блок располагается отдельно от заголовка (то есть ниже по тексту программы). Такое описание подпрограммы называется *опережающим* или *предварительным.* Перед блоком подпрограммы ее заголовок повторяется в полной или сокращенной форме – без списков параметров и

РЕПОЗИТОРИЙ БГПУ

типа результата для функций.

П р и м е р :

```
procedure P( x : real); FORWARD;
procedure Q( x : byte);
begin
   ... P(a)...  { вызов встречается до полного определения процедуры P }
end;
procedure P; { список формальных параметров можно не повторять }
begin
   ... Q(b) ...
end;
```

Опережающее описание может значительно улучшить восприятие большой программы, когда сначала располагаются все заголовки ее подпрограмм, а затем – сами подпрограммы.

Побочный эффект при вычислении функций

Любая функция дает некоторое значение (результат). Если в ее описании встречаются присваивания нелокальным переменным или параметрам-переменным, то помимо своего прямого назначения, функция как подпрограмма производит дополнительные алгоритмические действия. В этом случае говорят, что она имеет *побочный эффект*.

Рассмотрим пример программы, в которой вычисляется факториал:

```
var k : word;
function Factorial(var  n :  word) : word;
   var fact : word;
   begin
      fact:= 1; repeat  fact:= fact*n;  n:= n-1 until n = 1;
      Factorial:= fact
   end;
begin
   k:= 3; Writeln(k+Factorial(k));
   k:= 3; Writeln(Factorial(k)+k)
end.
```

Программа выводит на экран значения *9* и *7*. Во втором операторе печати сделана перестановка слагаемых в выражении, и сумма изменилась, т.к. при вычислении *Factorial(k)* переменная *k* получила новое значение *1*. Этот побочный эффект можно устранить, если в заголовке функции перед формальным параметром *n* убрать служебное слово ***var*** или использовать для ее вычисления вспомогательную локальную переменную.

Побочный эффект функций в программах обычно считается

РЕПОЗИТОРИЙ БГПУ

нежелательным.

Присваивание начальных значений

Турбо Паскаль разрешает присваивать *начальные значения* переменным любых типов, кроме файлов. Описание переменной с начальным значением записывается в *разделе описаний констант* в виде:

переменная : тип = список

Здесь *переменная* – это идентификатор, *тип* – идентификатор или изображение типа. Форма записи *списка* зависит от типа переменной. Для переменных простых типов этот список состоит из одного *константного выражения.* Правила записи списков для переменных структурных типов ясны из поясняющих примеров.

П р и м е р ы переменных с начальными значениями:

```
const
   LineLength : byte = 80;
   Radius : real = 1.0;
   CtrM : char = ^M;
   Pnt : pointer = nil;
   YesNo : Boolean = TRUE;
   AlphaBet : string[32] = 'абвгдежзийклмнопрстуфхцчшщьыъэюя';
   Separators : set of char = ['.',',',';',#9,#10,#13,#32];
   Digits : array[0..5] of char = ('0','1','2','3','4','5');
type
   Vector = array[1..2] of real;
   Matrix = array[1..2] of Vector;
const
   v : Vector = (1.0,1.0);
   m : Matrix = ((0.0,0.0), (1.0,1.0));
   r : record  x : byte;  v : Vector end = (x : 1.0; v : (1.0,1.0));
```

В программах переменные с начальными значениями рекомендуется использовать как константы структурных типов.

Эквивалентность и совместимость типов

В Турбо Паскале эквивалентность типов требуется в случае подстановки фактических параметров на место формальных в вызовах процедур и функций.

Типы *T1* и *T2* называются *эквивалентными*, если *T2* описан с помощью

РЕПОЗИТОРИЙ БГПУ

равенства *T2 = T1* или цепочки таких равенств.

П р и м е р :

```
type
   T1 = array[1..2] of real; T2 = T1;
var
   u : T1;  v : T2;  x : array[1..2] of real;
```

Переменные *u* и *v* принадлежат эквивалентным типам (и поэтому совместимы по присваиванию). Переменные *x* и *u* принадлежат разным типам, т.к каждое отдельное изображение структурного типа обозначает новый тип.

Совместимость типов требуется в арифметических, логических и строковых выражениях.

Типы *T1* и *T2 совместимы*, если:

1. *T1* и *T2* эквивалентны
2. *T1* и *T2* – типы целых или вещественных значений
3. *T2* – диапазон *T1,* либо *T1* и *T2* – диапазоны одного базового типа
4. *T1* и *T2* – типы множеств, а их базовые типы совместимы
5. *T1* – тип символьных строк*, T2* – тип символьных строк, либо символьный тип
6. *T1* – тип ***pointer***, *T2* – любой тип указателей
7. *T1* и *T2* – типы процедур с одинаковым числом параметров, типы которых эквивалентны
8. *T1* и *T2* – типы функций с одинаковым числом параметров, типы которых эквивалентны, и эквивалентны типы результата

Совместимость по присваиванию. Тип *T1* переменной в левой части оператора присваивания должен быть совместим с типом *T2* выражения в правой его части. Значение типа *T2* при этом не может выходить за границы возможных значений типа *T1*.

Тип *T2 совместим по присваиванию* с типом *T1* если:

1. *T1* и *T2* эквивалентны
2. *T1* и *T2*– типы вещественных значений
3. *T1* – тип вещественных значений, *T2* – тип целых значений
4. *T1* и *T2* – совместимые порядковые типы
5. *T1* и *T2*– типы строк
6. *T1*– тип строк, *T2* – символьный тип
7. *T1* и *T2* – совместимые типы множеств
8. *T1* и *T2* – совместимые типы указателей

РЕПОЗИТОРИЙ БГПУ

9. *T1 и T2* – совместимые типы подпрограмм

Функции приведения типов

В Турбо Паскале можно преобразовать переменную одного типа в переменную некоторого другого типа.

Пусть *x* – переменная, *T* – идентификатор типа. Для преобразования *x* к типу *T* используется специальная функция *T(x)*. Говорят, что *x приводится* к типу *T*.

Пример:

type
T = ***array****[1..2]* ***of byte****;*
numb = (One, Two, Three);
var
b : ***Boolean****; x :* ***integer****; c :* ***char****; r :* ***real****; a : T; m :* ***byte****; n : numb;*

В пределах действия этих описаний возможны присваивания:

char*(x):= 'A'; { эквивалентно x:= 65 }*
c:= ***char****(65); { эквивалентно c:= 'A' }*
b:= ***Boolean****(0); { эквивалентно b:=* ***false*** *}*
x:= ***integer****('A'); { эквивалентно x:= 65 }*
a:= T(x); { эквивалентно a[1]:= 65; a[2]:= 0; }
T(x)[1]:= 2; T(x)[2]:= 0; { эквивалентно x:= 2 }
x:= ***integer****(a); { эквивалентно x:= 65 }*
m:= ***byte****(Three); { эквивалентно m:= 2 }*
numb(m):= One; { эквивалентно m:= 0 }
n:= numb(1); { эквивалентно n:= Two }

Типы левой и правой частей в операторах присваивания должны иметь одинаковый размер в байтах для своих значений.

Замечание. Функции приведения типов *T(x)*, в отличие от функций преобразования *Ord, Chr, Trunc* или *Round,* не содержат никаких реальных вычислений над *x,* они предназначены только для отключения контроля типов.

Функция SizeOf, процедуры Move и FillChar

Функция *SizeOf* и процедуры *Move* и *FillChar* принадлежат стандартному модулю *System.*

Пусть *x* – переменная любого типа или идентификатор типа.

Функция *SizeOf(x)* дает размер *x* в байтах.

РЕПОЗИТОРИЙ БГПУ

П р и м е р :

```
type
   TArray = array[1..3] of byte;  YesNo = (Yes, No);
var
   m : TArray;  x : YesNo;
```

выражение	результат	выражение	результат
*SizeOf(**integer**)*	*2*	*SizeOf(TArray)*	*3*
*SizeOf(**pointer**)*	*4*	*SizeOf(m[3])*	*1*
SizeOf(m)	*3*	*SizeOf(x)*	*1*

Пусть *x1, x2* – переменные любых типов, *n* – выражение целого типа.

Процедура *Move(x1, x2, n)* копирует *n* смежных байтов из *x1* в *x2*. Ошибки не регистрируются.

П р и м е р :

```
var
   x1 : integer; x2 : array[1..4] of byte;
```

Оператор *Move(x1, x2, SizeOf(x1))* копирует целое значение *x1* в массив *x2*.

Пусть *x* – переменная любого типа, *n* – выражение целого типа, *v* – значение любого порядкового типа.

Процедура *FillChar(x, n, v)* заполняет *n* смежных байтов *x* значением младшего байта *v*.

П р и м е р :

```
var
   x : array[1..10] of real;
```

Оператор *FillChar(x, SizeOf(x), 0)* заполняет массив *x* нулевым значением.

Модули

Наборы констант, типов, переменных и подпрограмм можно объединить в транслируемые отдельно друг от друга *модули.* Каждый модуль – это *библиотека описаний*, которая может быть доступна программам и другим модулям.

Модули значительно ускоряют и упрощают разработку больших и сложных для понимания программ, особенно когда в ней участвуют несколько человек.

Модуль на языке Турбо Паскаль состоит из *заголовка, раздела интерфейса, раздела реализации* и необязательного *раздела инициализации.* В конце мо-

РЕПОЗИТОРИЙ БГПУ

дуля ставится точка.

В общем случае модуль записывается в виде:

unit *имя;*
interface *{ начало раздела интерфейса }*
uses *список модулей;*
разделы описаний констант, типов, переменных,
заголовки подпрограмм
implementation *{ начало раздела реализации }*
uses *список модулей;*
разделы описаний констант, типов,
переменных и подпрограмм
begin *{ начало раздела инициализации }*
список операторов
end.

Заголовок включает служебное слово ***unit*** и *имя* модуля – идентификатор. Все модули, используемые совместно, должны иметь различные имена.

Раздел интерфейса начинается со служебного слова ***interface*** и содержит описания констант, типов, переменных, а также заголовки процедур и функций, которые доступны (*видимы*) из программ и других модулей, использующих этот модуль. В списке после служебного слова ***uses*** перечисляются модули, константы, типы, переменные, процедуры и функции которых, в свою очередь, доступны (видимы) в этом модуле. При этом возможны случаи *косвенной* ссылки модулей, когда, например, модуль *M1* использует *M2,* а *M2* использует *M3.* Два модуля не могут ссылаться друг на друга в своих разделах интерфейса.

Раздел реализации начинается со служебного слова ***implementation*** и содержит полные описания процедур и функций из раздела интерфейса модуля. При этом их заголовки могут быть записаны в сокращенной форме – без списков параметров и типа результата для функций. Кроме того, раздел реализации может включать свои собственные описания констант, типов, переменных, процедур и функций, которые доступны только внутри этого модуля. Они недоступны (*скрыты*) для других модулей и программ. В списке после служебного слова ***uses*** перечисляются дополнительные модули, описания которых доступны (видимы) в разделе реализации этого модуля. Два модуля могут ссылаться друг на друга в своих разделах реализации.

Раздел инициализации, который начинается со служебного слова ***begin*** – это составной оператор, выполняющий любые алгоритмические действия. Здесь можно присвоить начальные значения переменным модуля или открыть нужные для работы файлы. При выполнении программы, использующей некоторый модуль, в первую очередь выполняются операторы раздела инициализа-

РЕПОЗИТОРИЙ БГПУ

ции этого модуля. Служебное слово ***end*** и точка в конце модуля обязательны.
П р и м е р самого простого модуля:

```
unit M;
interface
implementation
end.
```

Модуль *M* не содержит никаких описаний или алгоритмических действий.

Каждый модуль имеет свою систему обозначений. Это означает, что для его констант, типов, переменных, процедур и функций можно выбирать любые идентификаторы, не заботясь о том, встречаются ли такие идентификаторы в других программах и модулях. Чтобы различать описанные в разных модулях, но используемые вместе одинаковые идентификаторы, они записываются в виде составных имен:

идентификатор модуля **.** *идентификатор*

П р и м е р :

```
program P;                      unit M;
uses M;                         interface
var x : integer;                var x : integer;
begin                           implementation
   x:= 2;                       begin
   Writeln(x, M.x)                 x:= 1
end.                            end.
```

Программа *P* содержит идентификатор *x* и использует в то же время такой же идентификатор из модуля *M,* который записан с помощью составного имени.

В модуле, содержащем пустой раздел реализации, можно собрать константы и типы, которые будут использоваться во многих программах.

П р и м е р :

```
unit Calendar;
interface
type
   Week = (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
   Work = Mon..Fri;
   Month = (Jan,Feb,Mar,Apr,May,June,July,Aug,Sept,Oct,Now,Desem);
   Day = 1..31;
   Year = word;
   Date = record dd : Day; mm : Month; yy : Year end;
implementation
```


РЕПОЗИТОРИЙ БГПУ

```
end.
```

Модуль *Calendar* содержит типы данных для описания понятий, связанных с датами.

Модули делают практичной разработку *библиотек подпрограмм* для их многократного использования в других программах.

П р и м е р модуля:

```
unit Cmplx;
{ библиотека подпрограмм для работы с комплексными числами }
interface
type Complex = record Re, Im : real end; { комплексное число }
procedure Init(var C : Complex; R, I : real); { новое число }
procedure Add(A, B : Complex; var Result : Complex); { сложение }
procedure Mult(A, B : Complex; var Result : Complex); { умножение }
procedure Wrt(C : Complex); { печать }
implementation
procedure Init; { новое комплексное число }
begin
   with C do begin
      Re:= R; Im:= I
   end
end;
procedure Add; { сложение комплексных чисел }
begin
   with Resul do begin
      Re:= A.Re + B.Re; Im:= A.Im + B.Im
   end
end;
procedure Mult; { умножение комплексных чисел }
begin
   with Resul do begin
      Re:= A.Re*B.Re - A.Im*B.Im;
      Im:= A.Re*B.Im + B.Re*A.Im
   end
end;
procedure Wrt; { печать комплексного числа }
begin
   with C do begin
      Write(Re:6:3);
      if Im = 0 then Exit;
      if Im > 0 then Write(' + ');
      Write(Im:6:3); Write('i');
```

РЕПОЗИТОРИЙ БГПУ

```
   end
end;
end.
```

Модуль *Cmplx* содержит описание нескольких операций (процедур) для работы с комплексными числами, которые представляются парами вещественных чисел. В программе, использующей модуль *Cmplx,* можно ссылаться на любые идентификаторы из его раздела интерфейса. Для работы с комплексными числами можно не знать деталей реализации операций над ними, скрытых в разделе реализации. Число таких операций при желании можно расширить непосредственно в программе *P.* Можно также изменить реализацию любой из операций.

П р и м е р :

```
program P;
uses Cmplx;
var C1,C2,C3 : Complex;
begin
   Init(C1,1,1); Init(C2,1,0); Add(C3,C1,C2); Wrt(C3)
end.
```

Замечание. Если результатом трансляции обычных программ являются наборы данных (файлы DOS) с именами **.exe,* то оттранслированные модули помещаются в наборы данных с именами **.tpu.* В Турбо Паскале принято, что модуль и набор данных, содержащий текст этого модуля, должны иметь одинаковые имена. Пусть, например, модуль *Cmplx* находится в наборе данных *Cmplx.pas* (тип файла может быть произвольным). Тогда результатом трансляции этого модуля будет набор данных *Cmplx.tpu.*

РЕПОЗИТОРИЙ БГПУ

Стандартные модули

В Турбо Паскале имеется большое число констант, типов, переменных и подпрограмм, собранных в несколько библиотечных модулей:

Модуль	Назначение
System	*Основная библиотека подпрограмм Турбо Паскаля*
Crt	*Библиотека подпрограмм для работы с экраном и клавиатурой*
Graph	*Библиотека графических подпрограмм*
Strings	*Библиотека подпрограмм для работы с ASCIIZ - строками*
Dos	*Использование возможностей операционной системы DOS*
WinDos	*Использование возможностей DOS и ASCIIZ-строк*
Printer	*Для вывода на устройство печати*
Overlay	*Для организации программ с перекрытиями*
Turbo3	*Для совместимости с программами на Turbo Pascal 3.0*
Graph3	*Для работы с графикой Turbo Pascal 3.0*

Обычно в программе используются только те модули, которые необхо-

димы. Модуль *System* автоматически подключается к любой оттранслированной программе, и его не надо указывать в разделе модулей ***uses***. Этот модуль содержит все процедуры и функции стандартного Паскаля, подпрограммы ввода/вывода, подпрограммы работы со строками, с динамической памятью и др. Большинство из них подробно описаны в этом справочнике. Другие стандартные модули в этой части справочника не рассматриваются.

Все стандартные модули, кроме *Graph, Graph3* и *Turbo3* хранятся в наборе данных *Turbo.tpl.*

Функции для работы с клавиатурой KeyPressed и ReadKey

Функции *KeyPressed* и *ReadKey* содержатся в стандартном модуле *CRT.*

Логическая функция *KeyPressed* позволяет определить, была ли нажата на клавиатуре какая-нибудь клавиша. Специальные клавиши *<Shift>*, *<Ctrl>*, *<Alt>*, *<CapsLock>*, *<NumLock>* и *<ScrollLock>* не имеют самостоятельного значения (служат для изменения режима работы клавиатуры), и поэтому не учитываются. *KeyPressed* дает значение *TRUE*, если была нажата клавиша, и *FALSE* – в противном случае.

Пример:

repeat until *KeyPressed*

Этот цикл повторяется до тех пор, пока не будет нажата какая-нибудь клавиша. Цикл не содержит алгоритмических действий и может использоваться для задержки выполнения других операторов программы.

Функция *ReadKey* дает символ нажатой клавиши, который поступает в память для промежуточного хранения – *буфер клавиатуры.*

Наряду с клавишами, которым соответствуют печатные символы, клавиатура имеет ряд специальных клавиш. К ним относятся клавиши редактирования, клавиши перемещения курсора, управляющие и функциональные клавиши.

Клавиша *<BackSpace>* посылает в буфер клавиатуры символ *#8,* клавишы *<Tab>, <Enter>* и *<Esc>* – соответственно символы *#9, #13* и *#27.*

Клавиши *<Shift>*, *<Ctrl>*, *<Alt>*, *<CapsLock>*, *<NumLock>* и *<ScrollLock>* не имеют символьных значений.

Остальные специальные клавиши, а также многие комбинации клавиш с *<Shift>*, *<Ctrl>* и *<Alt>*, посылают в буфер клавиатуры два символа, первый из которых – символ *#0,* а второй – один из символов стандарта ASCII или IBM PC. Поэтому, если от функции *ReadKey* поступил пустой символ *#0,* то это означает, что была нажата какая-либо специальная клавиша или комбинация с

РЕПОЗИТОРИЙ БГПУ

<Shift>, *<Ctrl>* или *<Alt>*. Повторный вызов *ReadKey* в этом случае дает второй символ из буфера клавиатуры.

Примеры:

```
uses CRT;
var ch : char;
begin
repeat  until KeyPressed;
ch:= ReadKey;
case ch of
   'a'  : Writeln('Нажата клавиша <A>');
   'A'  : Writeln('Нажаты клавиши <Shift> и <A>');
   ' '  : Writeln('Нажата клавиша <Пробел>');
   #13 : Writeln('Нажата клавиша <Ввод>');
   #27 : Writeln('Нажата клавиша <Esc>');
   #0  :  { буфер клавиатуры содержит еще один символ }
   begin
      ch:= ReadKey;
      case ch of
        'A' : Writeln('Нажата клавиша <F7>');
        'H' : Writeln('Нажата клавиша <ВВЕРХ>')
      end
   end
end.
```

Цикл задерживает выполнение других операторов этой программы до нажатия на какую-нибудь клавишу. Присваивание дает символ от клавиши из буфера клавиатуры. Этот символ используется в операторах ***case*** для выбора сообщения и вывода его на экран.

```
function GetKeyChar : char;
   var key : char;
begin
   repeat  until KeyPressed;
   key:= ReadKey;
   if  key = #0  then key:= ReadKey;
   GetKeyChar:= key
end;
var  ch : char;
begin
   repeat
      ch:= GetKeyChar; Writeln(ch, Ord(ch) : 5)
   until ch = #27; {ESC}
end.
```

РЕПОЗИТОРИЙ БГПУ

В этой программе определена функция *GetKeyChar,* которая дает непустой символ из буфера клавиатуры. Функция используется для вывода на экран символов и их порядковых номеров, которые соответствуют нажатым клавишам. Программа завершает работу, если будет нажата клавиша *<Esc>*.

В следующих таблицах указаны непустые символы, которые соответствуют специальным клавишам и комбинациям с *<Shift>*, *<Ctrl>* и *<Alt>*.

Таблица соответствия для функциональных клавиш:

Нажата клавиша	**Символ**	**Вместе с <Shift>**	**Вместе с <Ctrl>**	**Вместе с <Alt>**
F1	*#59 ;*	*#84 T*	*#94 ^*	*#104 h*
F2	*#60 <*	*#85 U*	*#95 _*	*#105 i*
F3	*#61 =*	*#86 V*	*#96 ‘*	*#106 j*
F4	*#62 >*	*#87 W*	*#97 a*	*#107 k*
F5	*#63 ?*	*#88 X*	*#98 b*	*#108 l*
F6	*#64 @*	*#89 Y*	*#99 c*	*#109 m*
F7	*#65 A*	*#90 Z*	*#100 d*	*#110 n*
F8	*#66 B*	*#91 [*	*#101 e*	*#111 o*
F9	*#67 C*	*#92 *	*#102 f*	*#112 p*
F10	*#68 D*	*#93]*	*#103 g*	*#113 q*

Таблица соответствия для других специальных клавиш:

Нажата клавиша	**Символ**	**Вместе с <Ctrl>**	**Нажата клавиша**	**Символ**	**Вместе с <Ctrl>**
Home	*#71 G*	*#119 w*	*Вниз*	*#80 P*	
Вверх	*#72 H*		*PgDn*	*#81 Q*	*#118 v*
PgUp	*#73 I*	*#132*	*Insert*	*#82 R*	
Влево	*#75 K*	*#115 s*	*Delete*	*#83 S*	
Вправо	*#77 M*	*#116 t*	*PrtScr*		*#114 r*
End	*#79 O*	*#117 u*			

Таблица соответствия для буквенных клавиш:

Нажата клавиша	**Символ**	**Вместе с <Shift>**	**Вместе с <Ctrl>**	**Вместе с <Alt>**
A	*#97 a*	*#65 A*	*#1*	*#30*
B	*#98 b*	*#66 B*	*#2*	*#48 0*
C	*#99 c*	*#67 C*	*#3*	*#46 .*
D	*#100 d*	*#68 D*	*#4*	*#32*
E	*#101 e*	*#69 E*	*#5*	*#18*
F	*#102 f*	*#70 F*	*#6*	*#33*
G	*#103 g*	*#71 G*	*#7*	*#34 “*
H	*#104 h*	*#72 H*	*#8*	*#35 #*
I	*#105 i*	*#73 I*	*#9*	*#23*
J	*#106 j*	*#74 J*	*#10*	*#36 $*

РЕПОЗИТОРИЙ БГПУ

Нажата клавиша	Символ	Вместе с <Shift>	Вместе с <Ctrl>	Вместе с <Alt>
K	*#107 k*	*#75 K*	*#11*	*#37 %*
L	*#108 l*	*#76 L*	*#12*	*#38 &*
M	*#109 m*	*#77 M*	*#13*	*#50 2*
N	*#110 n*	*#78 N*	*#14*	*#49 1*
O	*#111 o*	*#79 O*	*#15*	*#24*
P	*#112 p*	*#80 P*	*#16*	*#25*
Q	*#113 q*	*#81 Q*	*#17*	*#16*
R	*#114 r*	*#82 R*	*#18*	*#19*
S	*#115 s*	*#83 S*	*#19*	*#31*
T	*#116 t*	*#84 T*	*#20*	*#20*
U	*#117 u*	*#85 U*	*#21*	*#22*
V	*#118 v*	*#86 V*	*#22*	*#47 /*
W	*#119 w*	*#87 W*	*#23*	*#17*
X	*#120 x*	*#88 X*	*#24*	*#45 -*
Y	*#121 y*	*#89 Y*	*#25*	*#21*
Z	*#122 z*	*#90 Z*	*#26*	*#44 ,*

Таблица соответствия для клавиш с цифрами и специальными символами:

<table>
<tr><th>Нажата клавиша</th><th>Символ</th><th>Вместе с <Shift></th><th>Вместе с <Alt></th></tr>
<tr><td></td><td>#48 0</td><td>#41)</td><td>#129</td></tr>
<tr><td>1</td><td>#49 1</td><td>#33 !</td><td>#120 x</td></tr>
<tr><td>2</td><td>#50 2</td><td>#64 @</td><td>#121 y</td></tr>
<tr><td>3</td><td>#51 3</td><td>#35 #</td><td>#122 z</td></tr>
<tr><td>4</td><td>#52 4</td><td>#36 $</td><td>#123 {</td></tr>
<tr><td>5</td><td>#53 5</td><td>#37 %</td><td>#124 |</td></tr>
<tr><td>6</td><td>#54 6</td><td>#94 ^</td><td>#125 }</td></tr>
<tr><td>7</td><td>#55 7</td><td>#38 &</td><td>#126 ~</td></tr>
<tr><td>8</td><td>#56 8</td><td>#42 *</td><td>#127 ⌂</td></tr>
<tr><td>9</td><td>#57 9</td><td>#40 (</td><td>#128</td></tr>
<tr><td>-</td><td>#45 -</td><td>#95 _</td><td>#130</td></tr>
<tr><td>=</td><td>#61 =</td><td>#43 +</td><td>#131</td></tr>
<tr><td>\</td><td>#92 \</td><td>#124 |</td><td></td></tr>
<tr><td>;</td><td>#59 ;</td><td>#58 :</td><td></td></tr>
<tr><td>‘</td><td>#39 ‘</td><td>#34 “</td><td></td></tr>
<tr><td>,</td><td>#44 ,</td><td>#60 <</td><td></td></tr>
<tr><td>.</td><td>#46 .</td><td>#62 ></td><td></td></tr>
<tr><td>/</td><td>#47 /</td><td>#63 ?</td><td></td></tr>
<tr><td>[</td><td>#91 [</td><td>#123 {</td><td></td></tr>
<tr><td>]</td><td>#93]</td><td>#125 }</td><td></td></tr>
</table>

РЕПОЗИТОРИЙ БГПУ

Литература

А. Авторские описания

1. Вирт Н. Язык программирования Паскаль. // Алгоритмы и организация решения экономических задач. М., 1974. Вып. 3. С. 38–66.
2. Вирт Н. Алгоритмы + структуры данных = программы. М., 1985.
3. Йенсен К., Вирт Н. Паскаль. Руководство для пользователя и описание языка. М., 1982.
4. Turbo Pascal 6.0. Руководство пользователя. Мн., 1992.

В. Учебники и пособия по языку Паскаль

1. Белецкий Я. Турбо Паскаль с графикой для персональных компьютеров. М., 1991.
2. Бородич Ю. С. Разработка программных систем на языке Паскаль. Мн., 1992
3. Епанишников А., Епанишников В. Программирование в среде Turbo Pascal 7.0. М., 1993.
4. Морозов А.А. Программирование на Turbo Pascal 7.0. Встроенные типы данных. Операторы. Мн., 1998.
5. Офицеров Д.В., Старых В.А. Программирование в интегрированной среде Турбо Паскаль. Мн., 1992.
6. Грогоно П. Программирование на языке Паскаль. М., 1982.

РЕПОЗИТОРИЙ БГПУ

Оглавление

Предисловие 3
Классификация типов в языке Turbo Pascal 7.0 4
Типы массивов 5
Индексированные переменные 6
Типы множеств 8
Типы записей 10
Оператор над записями with 12
Записи с вариантами 12
Типы указателей 14
Динамические переменные 15
Тип указателей pointer 17
Типы файлов 18
Операции над файлами 19
Текстовые файлы 21
Операции над текстовыми файлами 22
Контроль операций ввода/вывода 25
Стандартные файлы Input, Output и Lst 27
Подпрограммы 27
Описание процедуры 28
Описание функции 29
Блочная структура программ 30
Типы подпрограмм 31
Подстановка параметров 32
Процедуры Halt и Exit 36
Рекурсивные алгоритмы и программы 36
Побочный эффект при вычислении функций 40
Присваивание начальных значений 41
Эквивалентность и совместимость типов 41
Функции приведения типов 43
Функция SizeOf, процедуры Move и FillChar 43
Модули 44
Стандартные модули 48
Функции для работы с клавиатурой KeyPressed и ReadKey 49
Литература 53

РЕПОЗИТОРИЙ БГПУ