Structure of the Program:

Program Program_name; *Uses* *Const* *Type* *Var*	- name of the program - modules (*uses*, *crt*,…) – description of constants – description of types – description of variables
Procedure Procedure_name{({Var} x:type)}; - *Var* *Begin* *<body of the procedure>* *End;*	Description of a procedure, which could include some arguments. - May include embedded (nested) procedures and functions - Is executed by calling its name
Function Function_name({Var} x:type1):type2 *Var* *Begin* *<body of the function>* *Function_name:=z;* *End;*	Description of a function, which could include some arguments. - type of the result MUST be specified - May include embedded (nested) procedures and functions. - A value MUST be assigned to the function - Function is executed by calling its name *or* - Its result is assigned to a variable

BEGIN
<body of the program>
END.
	Main program body

Global and Local variables. Arguments for procedures and functions.

Procedures and functions operate with *local* and *global* variables. The values assigned to variables can be changed. Local variables are available only to the procedure (function) in which they are declared, or to embedded procedures (functions)

Procedure P1;
Function F1:type;
	- no arguments used
	- such procedures (functions) always do the same operations applied to the variables listed in the procedure (function) body

Procedure P2(x:type);
Function F2(x:type1):type2;
	- value-parameter(s) is (are) used as the argument(s) of the procedure (function)
	- the procedure (function) uses this value, but does not change the variable to which this value is assigned

Procedure P3(Var x:type);
Function F3(Var x:type1):type2;
	- variable-parameter(s) is (are) used as the argument(s) of the procedure (function)
	- the procedure (function) uses this value and is able to change the variable to which this value is assigned

Syntactic constructions

```
While <condition>
    do
     Begin
        <loop body>
     End;
```

```
Repeat
   Begin
      < loop body >
   End
  Until < condition >;
```

```
For i:=x1 to (downto) x2 do
      Begin
           < loop body >
      End;
```

- loop can be embedded into the parent loop
- condition can be compound (complex), i.e. it can include simple conditions combined by the logic operators

```
Case X of
   X1: Begin <…> End;
   X2: Begin <…> End;
   …
   Xn: Begin <…> End;
```

Case (choice) operator:
In case if X =
X1 – <list of operators>
…

String data type

```
Var    s:string; {length = 255}
       s1: string [20];
       s2: array [1..n] of string;
```

Functions:

length(s);
copy(s,n,m) – from *s*, *m* elements, starting from *n*
pos(s1,s2) – finds if *s1* is included in *s2*, returns position of the first inclusion
concat(s1,s2) - concatenation
read(s), *write(s)* – reading and writing
read(f,s), write(f,s) – reading (writing) from (to) file

Procedures:

Delete(s,n,m) – from *s*, *m* elements, starting from *n*
Insert(s1,s2,n) – inserts *s1* into *s2*, starting from *n*
Str(n,s) – converts number *n* into string *s*
Val(s,n,k) – converts string *s* into number *n, k* – index of the conversion error in case of erratic conversion, *k=0* if the conversion was correct

Symbol data type:

```
 Var c:char;
```

chr(*x*) – returns a symbol corresponding to the code *x*.
ord(*ch*) – returns a code corresponding to the symbol *ch*
Pred(cp) – returns the preceding symbol
Succ(ch)r – returns the next symbol

Recurrence – function or procedure addressing itself:

```
Function Factorial(n:integer):longint;
  Var i:byte;
  Begin
    if (n=0) or (n=1) then Factorial:=1
     else Factorial:=n*Factorial(n-1);
  End;
```

Debugging the program:

Debug → Add Watch <list variables which values you want to monitor during the debugging> → Watches

Execute the program step-by-step:

F7 – with access to the procedures and functions,
F8 – without access to the procedures and functions

One-dimensional arrays. Searching the array. Sorting the elements of the array.

One-dimensional array – a finite set of elements, each of them has its one value and position: (A = [a_1, a_2,..a_i,..a_n]). Each element can be addressed by the array name and position of that element.

Declaring of an array of n elements of the same type:

<table>
<tr><td rowspan="2">Type MyArray = Array [1..10] of Integer;
Type DArray = Array [1..n, 1..m] of Integer;

Var A: MyArray; Var A1: DArray;

- Type – syntax word;
- MyArray –name of the array;
- Description of the array;</td><td>Var B: array [1..10] of Integer;
Var B1: array [1..n, 1..m];</td></tr>
<tr><td>Const D: array [1..10] of integer = (a[1], a[2],…a[n])</td></tr>
<tr><td colspan="2">In order to output an array (on the screen or into a file) usually a loop is used, organized as a procedure (function):</td></tr>
<tr><td>Procedure Print(A:mas);
Var i:byte;
Begin
For i:=1 to n do write(a[i]:2:2,’ ‘);
End;</td><td>write(x:2:2); - formatted output, where number of integer and fractional digits is fixed.</td></tr>
</table>

Methods of forming an array:

1) From the keyboard: *for i:=1 to n do ReadLn(a[i]);*
2) Reading from a file: *for i:=1 to n do Read(f, a[i]);* here *f* is a file variable of text type, to which the file of interest is assigned:

   f:text;

   To “talk” to the file:

   1) Connect the file to the file variable: *assign(f, ‘filename.txt’);*
   2) Open file for reading/writing/adding:
      - *reset(f);* - open for reading data from the file
      - *rewrite(f);* – open for writing data into the file
      - *append(f);* - open for adding data into the file
   3) Close the file: *close(f)*
   4) Logic functions, which determine end of the file / end of the line: *eof(f), eoln(f)*

3) Random array:

   Randomize;

   for i:=1 to n do a[i]:=x0+(Random(x1)); (a:=Random(x1) ⇔ a:=X, $0 \leq X < x1$).

Searching the arrays. Sorting the elements of the array. *Efficiency of the sorting algorithms.*

Searching in an ordered (presorted) array by the bisection method.

Simple methods of sorting an array.

1) Simple swap (bubble method)

- look through the array of *n* elements, if a[i] > (<) a[i+1], then swap their positions;
- look through the array of *n-1* elements: i:=1..n-1;

Number of comparisons *N-i at each i step, total number of steps – N-1, hence complexity of the algorithm is* $C = N*(N-1)/2 \Rightarrow C = O(N^2)$

2) Selection sort.

- find max (min) element of the array, swap its position with the position of the first (last) element of the array, now the max (min) element is on its position;
- find next max (min) element, put it on its position;

Number of comparisons : *1-st run – N-1, 2-nd run – N-2,…,hence complexity of the algorithm is* $C = N\text{-}1 + N\text{-}2 + \ldots + 1 = N*(N-1)/2 \Rightarrow C = O(N^2)$

3) Insertion sort.

- assume, that a part of the array containing *i-1* elements at the i-step is presorted;
- take element on the *i-position* and put it on its position in the presorted part of the array;

complexity of the algorithm is $C = N*(N-1)/2 \Rightarrow C = O(N^2)$

Methods of fast sorting

1) Sort by merging
2) "Fast sort" of Hoare (Hoare, 1960)
3) Heap sort.

Some standard functions

Function	Result
sqr(x)	x^2
sqrt(x)	$x^{1/2}$
sin(x), cos(x), arctan(x)	
abs(x)	$\|x\|$
exp(x)	e^x
int(x)	integer part of the value
frac(x)	fractional part of the value
round(x)	rounding the number
trunc(x)	truncating the fraction
a mod b	residue from division (division remainder)
a div b	integer division (quotient)

Some standard types of data

Name of the type (Identifier)	Size (bytes)	Value range
Integer types		
Byte	1	0..255
Shortint	1	-128..127
Integer	2	-32768..32767
Word	2	0..65535
Longint	4	-2147483648..2147483647
Real types		
Real	6	$2.9\times10^{-39} – 1.7\times10^{38}$
Single	4	$1.5\times10^{-45} – 3.4\times10^{38}$
Double	8	$5\times10^{-324} – 1.7\times10^{308}$
Extended	10	$3.4\times10^{-4932} – 1.1\times10^{4932}$
Logic type		
Boolean	1	{true; false}
Symbol type		
Char	1	all symbols presented in ASCII