

Κώστας Γρ. Συμεωνίδης

begin - end.

Θεσσαλονίκη 1996

Κάθε γνήσιο αντίτυπο έχει την υπογραφή του συγγραφέα.

Κώστας Γρ. Συμεωνίδης
ksymeon@hyper.gr

©1996 Κώστας Συμεωνίδης

Συγκεκριμένα τμήματα ©1988-1993 Κώστας Συμεωνίδης

Απαγορεύεται η αναπαραγωγή οποιουδήποτε τμήματος του βιβλίου με οποιοδήποτε μέσο (φωτοτυπία, εκτύπωση, μικροφίλμ ή άλλη μηχανική ή ηλεκτρονική μέθοδο) χωρίς την έγγραφη άδεια του συγγραφέα. Εξαίρεση αποτελούν τα κομμάτια του κώδικα σε Pascal του βιβλίου τα οποία μπορούν να γραφτούν και να εκτελεστούν σε κάποιο πρόγραμμα υπολογιστή, αλλά απαγορεύεται να αναπαραχθούν για άλλη έκδοση.

...σε όλα εκείνα τα παιδιά που υπήρξαν μαθητές μου
και μου έδωσαν τη χαρά να είμαι δάσκαλος τους.

Κώστας Συμεωνίδης

ΠΕΡΙΕΧΟΜΕΝΑ

ΠΕΡΙΕΧΟΜΕΝΑ	iv
ΠΡΟΛΟΓΟΣ	xi
ΕΥΧΑΡΙΣΤΙΕΣ (CREDITS)	xiii

ΜΕΡΟΣ Ι - Η γλώσσα Turbo Pascal

ΚΕΦΑΛΑΙΟ 0	2
Ιστορία της Pascal (για να μη ξεχνιόμαστε...)	2
ΚΕΦΑΛΑΙΟ 1	4
Τι είναι ένα πρόγραμμα της Pascal;	4
Ενα πρόγραμμα της Turbo Pascal.....	4
Procedures και Functions	4
Statements (Προτάσεις)	5
Expressions (Εκφράσεις).....	5
Types, Variables, Constants και Typed Constants	6
Βάζοντας τα όλα μαζί	6
Input - Output	7
write και writeln (έκφραση, ...);	7
readln (μεταβλητή,);	8
ΚΕΦΑΛΑΙΟ 2	10
Declarations (Δηλώσεις)	10
Δηλώσεις Uses	10
Δηλώσεις Μεταβλητών (Variables)	11
Δηλώσεις Σταθερών (Constants).....	11
Constant Expressions	12
Δηλώσεις Typed Constants	12
ΚΕΦΑΛΑΙΟ 3	14
Data Types (Τύποι Δεδομένων)	14
Simple Types	14
Simple: Ordinal Types	15
Simple: Ordinal: Integer Types	15
Simple: Ordinal: Boolean Types	16
Simple: Ordinal: Char Type	17
Simple: Ordinal: Enumerated Types.....	18
Simple: Real Types	20
String Types	20
Σχέση String και Char.....	21
Type Compatibility (Συμβατότητα Τύπων Δεδομένων)	21
ΚΕΦΑΛΑΙΟ 4	22
Expressions και Operators.....	22
Operators (Τελεστές).....	22
Arithmetic Operators (Αριθμητικοί Τελεστές)	22
Boolean Operators (Λογικοί Τελεστές)	23

String Operator.....	23
Set Operators (Τελεστές Συνόλων)	23
Relational Operators (Τελεστές Σύγκρισης).....	23
ΚΕΦΑΛΑΙΟ 5	25
Statements (Προτάσεις).....	25
Simple Statements (Απλές Προτάσεις).....	25
Assignment Statements (Προτάσεις Αντικατάστασης)	25
Procedure Statements (Κλήση Διαδικασιών).....	25
Structured Statements (Δομημένες Προτάσεις)	26
Compound Statements (Σύνθετες Προτάσεις)	26
Conditional Statements (Υποθετικές Προτάσεις)	26
Repetitive Statements (Επαναληπτικές Προτάσεις).....	28
With Statements.....	29
ΚΕΦΑΛΑΙΟ 6	30
Procedures και Functions	30
Procedure Declarations (Δηλώσεις Διαδικασιών)	30
Function Declarations (Δηλώσεις Συναρτήσεων).....	32
Local Variables (Τοπικές Μεταβλητές).....	33
Ορισμός Παραμέτρων	33
Value Parameters.....	34
Constant Parameters (Παράμετροι Const)	34
Variable Parameters (Παράμετροι Var)	35
ΚΕΦΑΛΑΙΟ 7	37
Strings.....	37
Procedures & Functions για Χειρισμό Strings.....	38
function Length (s : string) : byte;.....	38
function Pos (substr,s : string) : byte;	38
function Copy (s : string; index,count : integer) : string;	38
function Concat (s1 [, s2, ..., sn] : string) : string;.....	38
procedure Insert (s1 : string; var s : string; index : integer);.....	39
procedure Delete (var s : string; index,count : integer);.....	39
ΠΑΡΑΤΗΡΗΣΕΙΣ	39
procedure Str (x [:width [:decimals]]; var s :string);.....	40
procedure Val (s : string; var v; var code : integer);	40
Περίληψη String Procedures & Functions.....	41
Functions	41
Procedures.....	41
Γενικά για τα Strings	41
ΚΕΦΑΛΑΙΟ 8	42
Arrays (Πίνακες)	42
Διαχείριση Πινάκων	43
Πρόβλημα.....	43
Πίνακες Δύο Διαστάσεων (Διδιάστατοι)	44
Πρόβλημα.....	46
Γενικά για τους Πίνακες	47
ΚΕΦΑΛΑΙΟ 9	48
Sets (Σύνολα)	48
Παραδείγματα Δηλώσεων και Χρήσης Sets.....	48
Πράξεις με Sets	48

Χρήση Συνόλων (Sets).....	49
procedure Exclude (var S : set of <u>type T</u> ; i : <u>type T</u>);.....	50
procedure Include (var S : set of <u>type T</u> ; i : <u>type T</u>);.....	50
ΚΕΦΑΛΑΙΟ 10	51
Records.....	51
Δηλώσεις Records.....	51
Χειρισμός Records - Πρόταση with.....	52
ΚΕΦΑΛΑΙΟ 11	54
Files (Αρχεία).....	54
Διαχωρισμός Αρχείων από το DOS	54
Διαχωρισμός Αρχείων λόγω Προσπέλασης τους.....	54
Τα Αρχεία στην Turbo Pascal.....	55
Text Files (Αρχεία Κειμένου).....	55
Χειρισμός των Text Files	56
Procedures και Functions για Text Files	57
procedure Assign (var f; Filename : string);	57
procedure Reset (var f);	57
procedure Rewrite (var f);	58
procedure Append (var f : text);	58
procedure Close (var f);.....	58
procedure Read (var f,);.....	58
procedure ReadLn (var f,);	58
procedure Write (var f,);.....	58
procedure WriteLn (var f,);.....	58
function EOF (var f) : boolean;.....	58
Παραδείγματα με Text Files.....	58
Δημιουργία ενός Text File.....	59
Πρόσθεση στοιχείων σε Text File	59
Εκτύπωση δεδομένων ενός Text File	59
Binary Files (Random Access)	60
Typed Binary Files.....	61
Οργάνωση και χειρισμός ενός Typed Binary File.....	62
Procedures και Functions για Typed Binary Files	63
procedure Assign (var f; Filename : string);	63
procedure Rewrite (var f);	63
procedure Reset (var f);	63
procedure Seek (var f; n : longint);.....	63
procedure Truncate (var f);.....	63
procedure Write (var f; <μεταβλητή τύπου δεδομένων του αρχείου>);	63
procedure Read (var f; <μεταβλητή τύπου δεδομένων του αρχείου>);	63
function FilePos (var f) : longint;.....	63
function FileSize (var f) : longint;.....	64
Παραδείγματα με Typed Binary Files.....	64
Installation Αρχείου (Ανεξάρτητο πρόγραμμα).....	64
Πρόγραμμα διαχείρισης του αρχείου.....	65
Επιλογή 1. Εισαγωγή Στοιχείων Μαθητή.....	66
Επιλογή 2. Διόρθωση Στοιχείων Μαθητή.....	67
Επιλογή 3. Διαγραφή Μαθητή.....	67
Επιλογή 4. Κατάλογος Μαθητών	68
Untyped Binary Files	69
Χειρισμός των Untyped Binary Files	70
Procedures και Functions για Untyped Binary Files.....	71
procedure Assign (var f; Filename : string);	71
procedure Rewrite (var f : file [; BlockSize : word]);.....	71

procedure Reset (var f : file [; BlockSize : word]);	71
procedure Seek (var f; n : longint);	71
procedure Truncate (var f);	72
procedure BlockRead (var f : file; var Buf; Count : word [; var Result : Word]);	72
procedure BlockWrite (var f : file; var Buf; Count : word [; var Result : Word]);	72
function FilePos (var f) : longint;	72
function FileSize (var f) : longint;	72
Παραδείγματα με Untyped Binary Files	73
Εγγραφή Δεδομένων σε Untyped Binary File	73
Εκτύπωση Στοιχείων Οποιοδήποτε Αρχείου	74
Αντιγραφή Αρχείου	74
Έλεγχος I/O Errors	75
ΚΕΦΑΛΑΙΟ 12	77
Δυναμικές μεταβλητές - Pointers	77
Γιατί Pointers;	77
Τεράστιες Ποσότητες Δεδομένων	77
Δεδομένα με Άγνωστο Μέγεθος	78
Δηλώσεις και Είδη Pointers	78
Δυναμικές Μεταβλητές	79
Procedures και Functions για Pointers	80
function MemAvail : longint;	80
function MaxAvail : longint;	80
procedure New (var p : pointer);	81
procedure GetMem (var p : pointer; Size : word);	81
procedure Dispose (var p : pointer);	81
procedure FreeMem (var p : pointer; Size : word);	81
function SizeOf (x) : word;	81
Η τιμή nil	81
Παραδείγματα με Pointers	82
Ελεύθερη μνήμη του Heap	82
Τεράστιοι πίνακες	82
ΚΕΦΑΛΑΙΟ 13	84
Units	84
Τι είναι Unit	84
Η Δομή ενός Unit	85
Επικεφαλίδα ενός Unit	85
Interface	86
Implementation	86
Κύριο Μέρος - Αρχικός Κώδικας	87
Παράδειγμα Unit	87
Μεγάλα Προγράμματα με Units	88
ΚΕΦΑΛΑΙΟ 14	89
Graphics - Γραφικά	89
Απαραίτητες Εντολές	89
Βασικές Έννοιες	90
Οι Σημαντικότερες Εντολές Γραφικών	90
procedure PutPixel (x,y : integer; c : color);	91
function GetPixel (x,y : integer) : word;	91
function GetMaxColor : word;	91
procedure SetColor (c : word);	91
procedure Line (x1,y1,x2,y2 : integer);	91
procedure MoveTo (x,y : integer);	91
procedure LineTo (x,y : integer);	91

procedure Rectangle (x1,y1,x2,y2 : integer);	91
procedure Circle (x,y,r : integer);	91
function GetMaxX : integer;	91
function GetMaxY : integer;	92
procedure ClearDevice;	92
Παραδείγματα με Graphics	92
Τυχαία Pixels	92
Animation Γραμμής	92

ΜΕΡΟΣ II - Object Oriented Programming

ΚΕΦΑΛΑΙΟ 0	95
Εισαγωγή στον OOP	95
Χαρακτηριστικά του OOP	96
Η Πρόκληση και η Δυσκολία	96
Αντικείμενα	97
ΚΕΦΑΛΑΙΟ 1	98
Encapsulation	98
Δηλώσεις Object	98
Το Νόημα του Encapsulation	99
Ενα Συγκριτικό Παράδειγμα	100
Τυχαίες γραμμές στην οθόνη	100
Συμπέρασμα	102
ΚΕΦΑΛΑΙΟ 2	103
Inheritance - Κληρονομικότητα	103
Δηλώσεις Descendant Objects (Απογόνων)	103
Κλήση Μεθόδων	104
Inherited (Κληρονομημένες) Μέθοδοι	105
Units με Αντικείμενα	106
Παράδειγμα με Inherited Objects	107
Unit με τα αντικείμενα Location και Pixel	107
ΚΕΦΑΛΑΙΟ 3	109
Polymorphism - Πολυμορφισμός	109
Κληρονομώντας Στατικές Μεθόδους	109
Virtual Methods	111
Constructors	113
Επεκτασιμότητα των Objects	114
Συμβατότητα Objects	115
ΚΕΦΑΛΑΙΟ 4	117
Δυναμικά Objects	117
Allocation και Initialization με τη New	117
Dispose για Δυναμικά Objects	118
Destructors	119
ΚΕΦΑΛΑΙΟ 5	121
OOP - Επίλογος	121
Και τώρα, τι κάνουμε;	121
ΠΑΡΑΡΤΗΜΑ Α	122

Library Reference..... 122

System Unit..... 122

function Abs (x : <u>integer ή real type</u>) : <u>ίδιο με το x</u> ;	122
function ArcTan (x : real) : real;	122
procedure Break;	122
procedure ChDir (s : string);	122
procedure Continue;	122
function Cos (x : real) : real;	123
procedure Dec (var x : <u>ordinal type</u> [; n : longint]);	123
procedure Erase (var f);	123
procedure Exit;	123
function Exp (x : real) : real;	124
procedure FillChar (var x; Count : word; Value : _);	124
function Frac (x : real) : real;	124
procedure GetDir (d : byte; var s : string);	124
procedure Halt [(ExitCode : word)];	124
procedure Inc (var x : <u>ordinal type</u> [; n : longint]);	124
function Int (x : real) : real;	125
function IOResult : integer;	125
function Ln (x : real) : real;	125
procedure Mkdir (s : string);	125
procedure Move (var Source, Dest; Count : word);	125
function Odd (x : longint) : boolean;	125
function Ord (x : <u>ordinal type</u>) : longint;	125
function ParamCount : word;	126
function ParamStr (Index : byte) : string;	126
function Pi : real;	126
function Pred (x : <u>ordinal type</u>) : <u>ίδιο type με του x</u> ;	126
function Random : real;	126
function Random (n : word) : word;	126
procedure Randomize;	127
procedure Rename (var f; Newname : string);	127
procedure Rmdir (s : string);	127
function Round (x : real) : longint;	127
function Sin (x : real) : real;	127
function SizeOf (x) : word;	127
function Sqr (x : <u>integer ή real type</u>) : <u>ίδιου τύπου με το x</u> ;	127
function Sqrt (x : real) : real;	127
function Succ (x : <u>ordinal type</u>) : <u>ίδιο type με του x</u> ;	128
function Trunc (x : real) : longint;	128
function UpCase (c : char) : char;	128

Crt Unit..... 128

var CheckBreak : boolean;	128
procedure ClrEol;	128
procedure ClrScr;	129
procedure Delay (ms : word);	129
procedure GotoXY (x,y : byte);	129
function KeyPressed : boolean;	129
function ReadKey : char;	129
procedure TextBackground (Color : byte);	130
procedure TextColor (Color : byte);	130
function WhereX : byte;	130
function WhereY : byte;	130

Dos Unit..... 130

function DiskFree (Drive : byte) : longint;	130
function DiskSize (Drive : byte) : longint;	130
procedure FindFirst (Path : string; Attr : word; var S : SearchRec);	130
procedure FindNext (var S : SearchRec);	131

procedure GetDate (var Year, Month, Day, DayOfWeek : word);	131
procedure GetTime (var Hour, Minute, Second, Sec100 : word);	131
record SearchRec	131
procedure SetDate (Year, Month, Day : word);	132
procedure SetTime (Hour, Minute, Second, Sec100 : word);	132
procedure UnpackTime (Time : longint; var DT : DateTime);	132
ΠΑΡΑΡΤΗΜΑ Β	133
ASCII Table	133
ΠΑΡΑΡΤΗΜΑ Γ	134
Κωδικοί Πλήκτρων της ReadKey	134
ΠΑΡΑΡΤΗΜΑ Δ	135
Reserved Words	135
ΠΑΡΑΡΤΗΜΑ Ε	136
Κανόνες Γραφής της Pascal	136
ΒΙΒΛΙΟΓΡΑΦΙΑ	139

Ενα μεγάλο μέρος του βιβλίου αυτού γράφτηκε σε μορφή σημειώσεων και μοιράστηκε σε σπουδαστές της Pascal στο χρονικό διάστημα από το 1988 μέχρι το 1994. Αρκετά κομμάτια των σημειώσεων αυτών αλλάχθηκαν και ενημερώθηκαν για να ενσωματωθούν σε αυτό το βιβλίο. Κάποια κεφάλαια γράφτηκαν από την αρχή.

Το βιβλίο αυτό προορίζεται για μία χρήση αναφοράς, όπου ο αναγνώστης μπορεί να ψάξει εύκολα και να βρει αυτό που τον ενδιαφέρει πλασιωμένο με παραδείγματα και λυμένες ασκήσεις. Είναι σχεδιασμένο για σπουδαστές - προγραμματιστές που ήδη προγραμματίζουν ή έστω κάνουν τα πρώτα τους δειλά βήματα στον προγραμματισμό. Είναι χωρισμένο σε κεφάλαια, τα οποία περιγράφουν το καθένα και μία πλευρά της Turbo Pascal. Εκτός από τα εισαγωγικά κεφάλαια που απλώς περιγράφουν κάποια στοιχεία της γλώσσας, έχει δοθεί ιδιαίτερη έμφαση στα κεφάλαια των strings, πινάκων, αρχείων και pointers όπου εκτός από απλή παράθεση σύνταξης και εντολών, υπάρχουν και παραδείγματα - λυμένες ασκήσεις. Η κατανόηση αυτών των στοιχείων της γλώσσας είναι ιδιαίτερα σημαντική για ανθρώπους που έχουν σκοπό να εμβαθύνουν στον προγραμματισμό.

Στο δεύτερο μέρος του βιβλίου καλύπτεται ένα μεγάλο μέρος της σύγχρονης Pascal, το Object Oriented Programming - OOP (Προγραμματισμός προσανατολισμένος στο αντικείμενο). Είναι ένα εντελώς καινούργιο στυλ προγραμματισμού, με μία μέθοδο που στην αρχή ξενίζει, γιατί ξεφεύγει από τη συνηθισμένη σκέψη. Επειδή ο προγραμματισμός στα νέα περιβάλλοντα εργασίας και λειτουργικά συστήματα (Windows, OS/2 κ.λ.π.) χρειάζεται απαραίτητα Objects, θεώρησα σκόπιμο να μην αφήσω έξω από το βιβλίο αυτό τον OOP.

Το βιβλίο αυτό δε μπορεί να διαβαστεί σαν μυθιστόρημα και δε μπορεί να αποτελέσει το μοναδικό εργαλείο για να μάθει κάποιος Pascal ή Turbo Pascal. Το θεωρώ όμως αναπόσπαστο κομμάτι της βιβλιοθήκης ενός σπουδαστή - προγραμματιστή και βάση για τη γνώση μιας από τις ομορφότερες (αν όχι η ομορφότερη) γλώσσες προγραμματισμού σήμερα.

Σε όλο το βιβλίο υπάρχει ιδιαίτερη χρήση της αγγλικής ορολογίας. Χωρίς να έχω τίποτα απολύτως με την Ελληνική γλώσσα, την οποία εκτιμώ βαθύτατα, δεν μπορώ να παραβλέψω το γεγονός ότι η επιστήμη των υπολογιστών έχει τους δικούς της όρους, στη γλώσσα που γεννήθηκε. Ετσι πολλές φορές αυτό που ξενίζει είναι η κακή μετάφραση

κάποιων όρων στα ελληνικά και όχι η χρήση των αυθεντικών όρων στα αγγλικά. Πολλοί έλληνες επειδή φοβούνται ότι η γλώσσα μας οδεύει προς την καταστροφή, εξελληνίζουν την ξένη ορολογία στον τομέα των υπολογιστών, πολλές φορές χρησιμοποιώντας λέξεις της δικής τους φαντασίας, χωρίς καμία απολύτως γνώση των υπολογιστών. Είναι όμως γνωστό ότι πολλά πράγματα δεν μεταφράζονται, ειδικά από απλούς φιλόλογους ή γραμματείς.

Ακόμα, πιστεύω ότι είναι το ίδιο δύσκολο (αν όχι δυσκολότερο) να μάθει κάποιος μία μεγάλη καινούργια "ελληνική" (;) λέξη χωρίς νόημα, από το να μάθει μία αγγλική λέξη η οποία μάλιστα χρησιμοποιείται και από όλον τον υπόλοιπο κόσμο. Για παράδειγμα η "μετάφραση" της λέξης pixel σε εικονοστοιχείο ή του CD σε οπτικοδισκόφωνο, μόνο γέλια μπορούν να προκαλέσουν παρά κατανόηση. Χρησιμοποιώντας λοιπόν αδόκιμα κακές ελληνικές μεταφράσεις - γλωσσοπλασίες, μένουμε πραγματικά "κλεισμένοι" στο κλουβί μας και δε νομίζω να το βλέπει κανείς αυτό ευχάριστα.

Όπως λοιπόν και οι ξένοι έχουν υιοθετήσει τις δικές μας ελληνικές λέξεις ορολογίας για πάρα πολλές επιστήμες (αστρονομία, ιατρική κ.α.) σεβόμενοι την πηγή των λέξεων αυτών, έτσι νομίζω ότι και εγώ πρέπει να σεβαστώ με τη σειρά μου τις λέξεις που γεννήθηκαν για τους υπολογιστές από κάποιους άλλους. Γιατί ας μην κρυβόμαστε. Μπορεί η Αρχαίοι Έλληνες να ήταν οι "πατέρες" πολλών και μεγάλων επιστημών, αλλά η σημερινή επιστήμη της πληροφορικής γεννήθηκε και μεγαλώνει κάπου αλλού. Ας είμαστε λοιπόν λιγότερο σωβινιστές και ας προσπαθήσουμε να συννενοηθούμε καλύτερα με τον έξω κόσμο.

ΕΥΧΑΡΙΣΤΙΕΣ (CREDITS)

Οι πρώτες ευχαριστίες πηγαίνουν σε όλους τους ανθρώπους που βοήθησαν στη συγγραφή, διόρθωση και εκτύπωση αυτού του βιβλίου. Ιδιαίτερα τη Βαγγελιώ Ασλάνογλου για την αμέριστη συμπαράστασή της στην έκδοση αυτή, την Αρτεμη Βουλγαρίδου και την Πόπη Τζιόβα για την ορθογραφική και συντακτική διόρθωση του κειμένου και τον φίλο και συνάδελφο Ζαφείρη Κεραμίδα για τις ουσιαστικές του προτάσεις και σχόλια στην ύλη του βιβλίου. Επίσης, ευχαριστώ το Data Station που μου έδωσε την ευκαιρία να διδάξω και να διαμορφώσω διδακτικά το αντικείμενο της Pascal, με τον τρόπο που εγώ ήθελα, πράγμα που στάθηκε η αιτία και η αφορμή της συγγραφής αυτού του βιβλίου.

Ευχαριστώ βέβαια και όλους τους μαθητές μου, που όλα αυτά τα χρόνια μου έδιναν πάντα την αστείρευτη χαρά να διδάσκω. Γιατί ξέρετε, η διδασκαλία είναι μία λειτουργία αμφίδρομη. Είναι ο δάσκαλος που συγκινεί το ακροατήριο, αλλά είναι και το ακροατήριο που προκαλεί το δάσκαλο να διδάξει σωστά. Το βιβλίο είναι αφιερωμένο σ' αυτούς που με την προσπάθειά τους, έκαναν κι εμένα να προσπαθήσω περισσότερο, για να τους δώσω τις τελειότερες και πιο σύγχρονες γνώσεις και τεχνικές στην επιστήμη της πληροφορικής.

Αν και είναι λίγο περίεργο θα ευχαριστήσω τον Niklaus Wirth για τη δημιουργία της Pascal καθώς και την Borland International που μας χάρισε αυτόν τον καταπληκτικό compiler - εργαλείο που λέγεται Turbo ή Borland ή Object Pascal και που έκανε πάρα πολλούς ανθρώπους, μαζί και εμένα, να αγαπήσουν τον προγραμματισμό.

Το βιβλίο αυτό γράφτηκε εξ' ολοκλήρου σε ηλεκτρονική μορφή. Η τελική του μορφή και τα κεφάλαια που προστέθηκαν στις υπάρχουσες σημειώσεις γράφτηκαν και σελιδοποιήθηκαν σε έναν υπολογιστή Intel 486@100MHz με το υπέροχο Microsoft Natural Keyboard. Το software που χρησιμοποιήθηκε για τη συγγραφή του βιβλίου ήταν το *Microsoft Word for Windows v6.0* που έτρεχε στα *Windows for Workgroups v3.11*. Το βιβλίο τυπώθηκε σε έναν Canon LBP-4 LaserPrinter, στα 300dpi. Η μακέτα του εξωφύλλου έγινε στο *CorelDraw v5.0* και τυπώθηκε σε έναν Epson Stylus Color InkJet Printer, στα 720dpi. Το στερεό που απεικονίζεται είναι ένα υπερβολικό δωδεκάεδρο φτιαγμένο στο *Mathematica v2.0 for Windows*.

ΜΕΡΟΣ Ι

Η γλώσσα Turbo Pascal

Ιστορία της Pascal (για να μη ξεχνιόμαστε...)

Στις αρχές της δεκαετίας του '70, ο καθηγητής Niklaus Wirth δημιούργησε την Pascal (ονομάστηκε έτσι προς τιμή του Blaise Pascal, μαθηματικό και φιλόσοφο του 17ου αιώνα). Η Pascal είναι μία γλώσσα προγραμματισμού, που δημιουργήθηκε με αρχικό σκοπό να είναι μία φυσιολογική - διαδικαστική (procedural) γλώσσα που θα βοηθήσει τους αρχάριους προγραμματιστές και σπουδαστές να αποκτήσουν "καλές" συνήθειες στον προγραμματισμό, δίνοντας τους τη δυνατότητα να γράψουν καθαρά, ευανάγνωστα, σωστά και δομημένα προγράμματα. Πριν από την Pascal οι σπουδαστές μάθαιναν τη FORTRAN μία πολύ παλιότερη και αδόμητη γλώσσα. Ο Niklaus Wirth όμως πίστευε ότι πολλά προγραμματιστικά λάθη, θα μπορούσαν να είχαν αποφευχθεί, αν στη θέση της FORTRAN υπήρχε μία γλώσσα δομημένη που είχε αυστηρό έλεγχο στους τύπους δεδομένων. Γρήγορα η Pascal έγινε μία γλώσσα αποδεκτή από την ακαδημαϊκή κοινότητα και άρχισε να διδάσκεται στα πανεπιστήμια, όχι μόνο σαν γλώσσα προγραμματισμού αλλά και σαν βάση για το μάθημα των δομών δεδομένων.

Από την παρουσίαση της το 1983 η Turbo Pascal έκανε μια επανάσταση στο χώρο των μικροϋπολογιστών. Η μεγάλη ταχύτητα της στο compilation, ο μοναδικός χώρος εργασίας που ενσωμάτωνε editor και compiler και οι πολύ βοηθητικές επεκτάσεις της standard Pascal την έκαναν αποδεκτή από εκατοντάδες χιλιάδες προγραμματιστές.

Από τότε η δημιουργός εταιρεία Borland International Inc. παρουσίασε πολλές εκδόσεις, κάθε μία με όλο και καλύτερες επεκτάσεις στη γλώσσα. Είναι αναμφισβήτητο ότι οι compilers της Borland είναι πλέον το παγκόσμιο standard στους Pascal compilers. Από την έκδοση 5.5 και μετά η Turbo Pascal διαθέτει και objects πράγμα που κάνει δυνατό τον Object Oriented Προγραμματισμό (OOP) που είναι και ότι πιο σύγχρονο στον τομέα του προγραμματισμού. Με βάση τις επεκτάσεις της γλώσσας σε objects δημιουργήθηκε και η Turbo Pascal for Windows που επέτρεπε στους προγραμματιστές να γράψουν προγράμματα που να τρέχουν κάτω από το πασίγνωστο περιβάλλον εργασίας της Microsoft.

Η τελευταία έκδοση της **Turbo Pascal** είναι η **v7.0** που παρουσιάστηκε το 1993, ενώ κυκλοφορούν ακόμη η **Turbo Pascal for Windows v1.5** και η **Borland Pascal with Objects v7.0** που συμπεριλαμβάνει compilers για DOS και για Windows.

Το καλοκαίρι του 1995 η Borland International Inc., παρουσίασε ένα ακόμη εργαλείο για τους προγραμματιστές με βάση την Pascal. Πρόκειται για την μετεξέλιξη της γλώσσας σε Visual Programming Language και ονομάζεται **Delphi**. Το Delphi είναι ουσιαστικά η όγδοη (8) έκδοση της Borland Pascal, που για λόγους πρακτικούς αλλά και εμπορικούς ονομάστηκε έτσι. Η “νέα” γλώσσα, η βάση του Delphi, που έχει αρκετές αλλαγές (τόσες που να θεωρείται κάτι εντελώς νέο) ονομάζεται πλέον Object Pascal.

Σήμερα η Pascal είναι μία πάρα πολύ δυνατή γλώσσα προγραμματισμού, παγκοσμίως αποδεκτή και με έναν από τους γρηγορότερους compilers από οποιαδήποτε άλλη γλώσσα (αν όχι ο γρηγορότερος). Οι τελευταίες αλλαγές - επεκτάσεις της γλώσσας που έγιναν στην έκδοση 7.0 και Delphi την έκαναν να αποκτήσει αρκετές ομοιότητες με τη γλώσσα C++, κρατώντας βέβαια την αυστηρότητα των δομών και των τύπων δεδομένων που έχει.

Τι είναι ένα πρόγραμμα της Pascal;

Τα επόμενα κεφάλαια ασχολούνται με τον ορισμό της γλώσσας Turbo Pascal. Κάθε Κεφάλαιο πραγματεύεται και ένα διαφορετικό στοιχείο - κομμάτι της γλώσσας. Όλα μαζί δημιουργούν ένα πρόγραμμα της Turbo Pascal. Το Κεφάλαιο αυτό δίνει μία σύντομη περιγραφή της γλώσσας και των στοιχείων που αποτελούν τα προγράμματα της.

Ενα πρόγραμμα της Turbo Pascal

Στην πιο απλή μορφή του ένα πρόγραμμα της Pascal αποτελείται από την επικεφαλίδα και το block του κυρίως προγράμματος. Το block αυτό καθορίζεται από δύο λέξεις: **begin** και **end**. Ανάμεσα σ'αυτές τις λέξεις υπάρχουν οι εντολές - προτάσεις του κυρίως προγράμματος. Παρακάτω είναι ένα πολύ απλό παράδειγμα προγράμματος σε Pascal.

```
program Welcome;  
begin  
    write ('Welcome to Turbo Pascal');  
end.
```

Η πρώτη γραμμή είναι και η επικεφαλίδα του προγράμματος η οποία στις εκδόσεις της Turbo Pascal είναι προαιρετική και άρα μπορεί να παραληφθεί. Το κυρίως πρόγραμμα στο παράδειγμα μας αποτελείται από μία μόνο πρόταση που τυπώνει στην οθόνη το μήνυμα Welcome to Turbo Pascal. Το κομμάτι αυτό θα μπορούσε να έχει και άλλες πολλές προτάσεις, αν και όπως θα δούμε παρακάτω όλοι οι σωστοί προγραμματιστές συνηθίζουν το κομμάτι αυτό να μη ξεπερνά τις 5-10 γραμμές!!!

Procedures και Functions

Ο κώδικας μεταξύ του begin και του end. σε ένα πρόγραμμα είναι η λογική του προγράμματος. Σε ένα πολύ μικρό πρόγραμμα είναι πολύ πιθανόν να μη χρειαστείτε τίποτε παραπάνω. Σε ένα μεγάλο όμως και σύνθετο πρόγραμμα ο περισσότερος κώδικας μας γράφεται στα “δομικά υλικά” του προγράμματος, τις procedures (διαδικασίες) και τις functions (συναρτήσεις). Η ουσία είναι να μπορέσουμε να “σπάσουμε” το πρόγραμμα μας σε πολλά μικρά κομμάτια που κάθε ένα να αποτελεί μία οντότητα που πραγματοποιεί μία χρήσιμη εργασία. Οι procedures και οι functions είναι ουσιαστικά τα εργαλεία του προγραμματιστή που τα φτιάχνει μόνος του για να τα χρησιμοποιήσει αργότερα. Μία

procedure ή μία function μοιάζουν πάρα πολύ με το κυρίως πρόγραμμα. Έχουν κι αυτές μία επικεφαλίδα και ένα “σώμα” με εντολές - προτάσεις, το οποίο περικλείεται ανάμεσα σε ένα begin - end;

```
procedure PrintFiveStars;  
begin  
    write ('*****');  
end;
```

Προσέξτε, ότι το end αυτό έχει στο τέλος ένα ελληνικό ερωτηματικό (;) ενώ το end του κυρίως προγράμματος έχει μία τελεία (.). Αυτό είναι και το μοναδικό σημείο ενός προγράμματος της Pascal που ένα end ακολουθείται από μία τελεία. Οι procedures και οι functions γράφονται μετά την επικεφαλίδα του προγράμματος και πριν από το κυρίως πρόγραμμα.

Statements (Προτάσεις)

Ο κώδικας της Pascal, οι εντολές όπως λέμε σε πολλές άλλες γλώσσες, είναι οι προτάσεις. Αυτές είναι που κάνουν όλη τη δουλειά σε ένα πρόγραμμα. Οι προτάσεις χωρίζονται σε απλές και σύνθετες. Μία απλή πρόταση μπορεί να δώσει μία τιμή, να ενεργοποιήσει μία procedure ή function κ.λ.π. Οι σύνθετες προτάσεις μπορούν να είναι υποθετικές, να επαναλαμβάνουν κάποιες άλλες υπο-προτάσεις κ.α. Μπορείτε να συγκρίνετε μία πρόταση της Pascal με μία πρόταση από κάθε “ανθρώπινη” γλώσσα. Μία πρόταση περιέχει μία ολόκληρη σκέψη. Οι σύνθετες προτάσεις απλά περιέχουν πιο σύνθετες σκέψεις...

Expressions (Εκφράσεις)

Μέσα σε μία πρόταση είναι πάρα πολύ πιθανό να συναντήσουμε κάποιες εκφράσεις. Οι εκφράσεις είναι κομμάτια που συνθέτουν μία πρόταση, αλλά δεν είναι σε καμία περίπτωση αυτόνομα. Π.χ. το 4+5 είναι μία έκφραση που δηλώνει μία πράξη αριθμητική (πρόσθεση), αλλά μόνη της δεν μπορεί να σταθεί. Συνήθως οι εκφράσεις είναι πράξεις αριθμητικές, λογικές και συγκρίσεις. Υπάρχουν και σύνθετες εκφράσεις οι οποίες απλά αποτελούνται από άλλες απλούστερες. Π.χ. $a > 4*(5+6)$

Types, Variables, Constants και Typed Constants

Οι μεταβλητές (variables) είναι χώροι αποθήκευσης που μπορούν να έχουν κάποια τιμή. Κάθε μεταβλητή όμως πρέπει να ανήκει σε κάποιο Τύπο Δεδομένων (Data Type). Ο τύπος μίας μεταβλητής, καθορίζει τον τρόπο αποθήκευσης των δεδομένων, τις τιμές που μπορεί να πάρει, καθώς και τις πράξεις που μπορούμε να κάνουμε μ' αυτήν. Στο παράδειγμα που ακολουθεί οι μεταβλητές X και Y είναι του τύπου integer, που σημαίνει ότι μπορούν να πάρουν μόνο ακέραιες αριθμητικές τιμές.

```
program Example;
const
  A  = 12;
  B  : integer = 23;
var
  X,Y : integer;
begin
  X := 7;
  Y := 8;
  X := Y+A;
  B := 57;
end.
```

Στο παραπάνω και λίγο “άχρηστο” πρόγραμμα, η μεταβλητή X παίρνει στη αρχή την τιμή 7, αλλά μετά από δύο προτάσεις αλλάζει και γίνεται το αποτέλεσμα της πρόσθεσης Y+A. Όπως βλέπετε, η τιμή μίας μεταβλητή μπορεί να μεταβάλλεται.

Το A είναι μία σταθερά (constant). Στην αρχή του προγράμματος δηλώνουμε ότι είναι ίση με 12 και αυτή η τιμή δεν μπορεί να αλλάξει σε όλη τη διάρκεια του προγράμματος.

Το B είναι μία σταθερά με τύπο δεδομένων (typed constant). Στην αρχή του προγράμματος τη δηλώνουμε ότι είναι ίση με 23, αλλά επίσης λέμε ότι είναι και του τύπου integer. Στη διάρκεια του προγράμματος αλλάζουμε αυτήν την τιμή. Ουσιαστικά μία typed constant είναι μία μεταβλητή η οποία έχει μία αρχική τιμή. Στην Pascal οι μεταβλητές μας, ΔΕΝ έχουν κάποιες σταθερές αρχικές τιμές. Η τιμή τους είναι άγνωστη. Π.χ. το πρόγραμμα

```
var
  X : integer;
begin
  write (X);
end.
```

που ουσιαστικά μας τυπώνει την τιμή της μεταβλητής X, δεν μπορούμε να ξέρουμε τι αποτέλεσμα θα βγάλει!!! Κάθε φορά που το τρέχουμε η τιμή του X θα μπορούσε να είναι διαφορετική. Εφόσον χρειάζεται, έχουμε την υποχρέωση στην Pascal να δίνουμε εμείς αρχικές τιμές στις μεταβλητές μας.

Βάζοντας τα όλα μαζί

Τώρα που ήδη γνωρίσατε τα βασικά στοιχεία ενός προγράμματος σε Pascal, μπορείτε να αρχίσετε να μαθαίνετε και με περισσότερη λεπτομέρεια τα στοιχεία που το συνθέτουν. Παρακάτω υπάρχει ένα παράδειγμα - σχεδιάγραμμα ενός προγράμματος γραμμένου σε Pascal.

<i>Επικεφαλίδα (προαιρετική)</i>	<code>program Example;</code>
<i>Δηλώσεις (Global)</i>	<code>const X = 10; var i : integer;</code>
<i>Procedures & Functions</i>	<code>procedure Star; begin write ('*'); end;</code>
<i>Κυρίως πρόγραμμα (main)</i>	<code>begin i := 5; Star; end.</code>

Input - Output

Σε κάθε γλώσσα προγραμματισμού, το πρώτο πράγμα που μαθαίνουμε είναι πως να τυπώνουμε διάφορα πράγματα, όπως επίσης και πως να "ζητάμε" κάποια πράγματα από το πληκτρολόγιο (από τον χρήστη). Στην Pascal υπάρχουν τέσσερις βασικές διαδικασίες. Οι **write** - **writeln** και οι **read** - **readln**.

- **write και writeln (έκφραση, ...);**

Με τις procedures αυτές, μπορούμε να τυπώνουμε κάποια πράγματα στην οθόνη. Είτε σταθερές, αριθμητικές ή αλφαριθμητικές, είτε τιμές μεταβλητών, είτε ακόμα και αποτελέσματα κάποιων πράξεων. Τα ξεχωριστά "πράγματα" που τυπώνουμε τα χωρίζουμε με κόμμα. Επίσης, όπως και σε όλες τις ρουτίνες της Pascal, αυτά που θέλουμε να τυπωθούν τα βάζουμε μέσα σε παρένθεση. Η διαφορά, τέλος, των **write** και **writeln** είναι ότι η **write** αφήνει τον cursor αμέσως μετά απ' αυτό που θα τυπώσει, ενώ η **writeln** τον μετακινεί στην αρχή της επόμενης γραμμής.

Δείτε το παρακάτω παράδειγμα :

```
var  
    s      : string;  
    a,b    : byte;  
begin  
    s := 'Κυριακή';  
    writeln (a, ' ', b);
```

Δίνω στο s την τιμή 'Κυριακή'.
Τυπώνω την τιμή του a, ένα κενό (space) και την
τιμή του b (τρία πράγματα συνολικά).

```

write ('Σήμερα είναι ');
writeln (s);
end.

```

Τυπώνω το μήνυμα 'Σήμερα είναι' ... ο cursor μένει ακριβώς δίπλα....
... και τέλος τυπώνω την τιμή του s (δηλαδή Κυριακή).

Στην Pascal μπορούμε να τυπώσουμε και με δεξιά στοίχιση (φορμαρισμένη εκτύπωση), πράγμα ιδιαίτερα χρήσιμο στους αριθμούς. Ετσι αν ζητήσουμε να τυπωθεί το `a:7`, σημαίνει ότι θέλουμε να τυπώσουμε την τιμή της μεταβλητής `a` χρησιμοποιώντας 7 χαρακτήρες. Αν το `a` είναι π.χ. 1000 (4 χαρακτήρες) τότε θα τυπωθούν 3 κενά και μετά το 1000. Σε περίπτωση που έχουμε έναν αριθμό τύπου `real` (δεκαδικός) μπορούμε να τυπώσουμε το `x:5:2`. Το `:5` σημαίνει ότι θέλουμε να τυπωθεί το `x` χρησιμοποιώντας 5 χαρακτήρες και το `:2` για να μας το τυπώσει με δύο δεκαδικά ψηφία. Ετσι αν το `x` είναι 3.421 θα τυπωθεί ' 3.42'.

Δείτε το παρακάτω παράδειγμα :

```

var
  a : real;
begin
  a := 1.25;
  writeln (a:7:3);

  writeln (123);
  writeln (12);
  writeln (1234);

  writeln (123:5);
  writeln (12:5);
  writeln (1234:5);
end.

```

Τυπώνεται :

```

~~1.250 (τα ~ είναι κενά)

123
12
1234
..... (σε πέντε χαρακτήρες)
  123
  12
 1234

```

- **readln (μεταβλητή,);**

Με την εντολή αυτή, μπορούμε να πάρουμε κάποιες τιμές από το πληκτρολόγιο. Είτε αριθμητικές, είτε αλφαριθμητικές. Η **readln** περιμένει να γράψουμε κάτι στο πληκτρολόγιο και μόλις πατηθεί το Enter το καταχωρεί στις μεταβλητές που της έχουμε δώσει. Παρόλο που η **readln** μπορεί να δεχθεί πολλές μεταβλητές, συνήθως τη χρησιμοποιούμε δίνοντας μόνο μία. Όταν διαβάζουμε νούμερα, παραλείπονται τα κενά. Όταν διαβάζουμε strings όλοι οι χαρακτήρες είναι επιτρεψίμοι. Στην Pascal υπάρχει και η εντολή **read**, αλλά δε χρησιμοποιείται για να πάρουμε τιμές από το πληκτρολόγιο (user input), αλλά για να διαβάζουμε από τα αρχεία. Περισσότερα δείτε στο παρακάτω παράδειγμα :

```

var
  a,b   : integer;
  s     : string;
begin
  readln (a);
  readln (a,b);

  readln (s);
end.

```

Διαβάζει την τιμή για τη μεταβλητή `a`.
Διαβάζει τιμές για τα `a` και `b`. Πρέπει να δοθούν χωρισμένες με ένα ή περισσότερα κενά (spaces).
Διαβάζει την τιμή για το string `s`. Δέχεται όλους τους χαρακτήρες. Μετρούν ΚΑΙ τα κενά.

Είναι χαρακτηριστικό ότι η `readln`, δε δείχνει τίποτα στην οθόνη, όταν ζητά κάποιες τιμές. Ένας `cursor` μόνο που αναβοσβήνει. Γι' αυτό είναι καλό να προηγείται πάντα μία `write` (προσέξτε `write` και όχι `writeln` για να μείνει ο `cursor` δίπλα!) η οποία να έχει κάποιο μήνυμα της μορφής "Δώσε έναν αριθμό : " ή κάτι τέτοιο. Η άνω - κάτω τελεία (:), όπως επίσης και τα κενά, καλό είναι να γράφονται για λόγους καλοτισμού.

Το παρακάτω πρόγραμμα, είναι ένα πλήρες παράδειγμα **input - output (I/O)** σε Pascal.

```
var
  a : integer;
begin
  writeln ('Πρόγραμμα υπολογισμού εμβαδού τετραγώνου');
  writeln;
  write ('Δώσε την πλευρά ενός τετραγώνου : ');
  readln (a);
  writeln ('Το εμβαδό του τετραγώνου είναι : ',a*2);
end.
```

Κ Ε Φ Α Λ Α Ι Ο 2

Declarations (Δηλώσεις)

Στο Κεφάλαιο αυτό περιγράφονται οι δηλώσεις *uses*, μεταβλητών, σταθερών και δικών μας τύπων δεδομένων.

Δηλώσεις Uses

Ενώ όλες οι υπόλοιπες δηλώσεις μπορούν να βρίσκονται σε όποιο σημείο του προγράμματος θεωρητικά θέλουμε (πρακτικά συνήθως γίνονται στην αρχή ή μέσα στις *procedures* και *functions* (τοπικές)) οι δηλώσεις *uses* μπορούν να μπουν μόνο στην αρχή του προγράμματος πριν από οτιδήποτε άλλο και ακριβώς μετά την επικεφαλίδα. Οι δηλώσεις αυτές χρησιμεύουν για να κοινοποιήσουμε στον *compiler* ποιά *units* θέλουμε να χρησιμοποιήσουμε στο πρόγραμμά μας. Ξεκινούν με τη λέξη ***uses*** και έχουν την εξής μορφή:

όνομα unit, ;

Π.χ.

```
uses  
  Crt,Dos;
```

Unit είναι μία βιβλιοθήκη με έτοιμες *procedures* και *functions*, αλλά ακόμα και με μεταβλητές, σταθερές και γενικά οτιδήποτε μπορεί να δηλωθεί στην *Pascal*. Συνήθως τα *units* είναι έτοιμα από κάποιους άλλους προγραμματιστές, αλλά μπορούμε να φτιάξουμε και δικά μας. Να θυμάστε πάντα ότι ένα *unit* έχει ήδη μεταφρασμένο κώδικα σε γλώσσα μηχανής οπότε το *compilation* του προγράμματάς μας δεν επιβαρύνεται και με *compilation* των ρουτινών ενός *unit*. Ο σκοπός που εξυπηρετούν τα *units* είναι ουσιαστικά να επεκτείνουν τη γλώσσα με διάφορες *procedures* και *functions* για το καλό του χρήστη.

Η *Turbo Pascal* έρχεται μαζί με μία έτοιμη και αρκετά μεγάλη συλλογή από *procedures* και *functions* που περιέχονται σε αρκετά *units*. Π.χ. υπάρχει το *unit Crt* το οποίο περιλαμβάνει ρουτίνες για χειρισμό πληκτρολογίου και οθόνης, το *Graph* το οποίο έχει εντολές γραφικών, το *Dos* που έχει γενικά εντολές του *dos* κ.α.

Δηλώσεις Μεταβλητών (Variables)

Στην Pascal είμαστε υποχρεωμένοι να δηλώσουμε τις μεταβλητές που θα χρησιμοποιήσουμε, είτε σε όλο το πρόγραμμά μας (global - καθολικές), είτε μέσα σε κάποια procedure ή function (local - τοπικές). Οι δηλώσεις μεταβλητών ξεκινούν με τη λέξη **var** και έχουν τη μορφή:

όνομα μεταβλητής. ... : τύπος δεδομένων της μεταβλητής;

Π.χ.

```
var
  X      : byte;
  a,b    : integer;
  s      : string[20];
```

Όπως φαίνεται από τα παραπάνω μπορούμε σε κάθε γραμμή να δηλώσουμε παραπάνω από μία μεταβλητή, αρκεί να χωρίσουμε τα ονόματά τους με κόμμα. Τα ονόματα των μεταβλητών μπορούν να περιέχουν γράμματα του λατινικού αλφαβήτου και αριθμητικά ψηφία, αλλά πρέπει οπωσδήποτε να αρχίζουν με γράμμα. Επίσης απαγορεύεται ρητά να έχουν κενό (!!) ή κάποιους άλλους χαρακτήρες που χρησιμοποιούνται για άλλο λόγο στην Pascal (π.χ. *, (, = κ.λ.π.) Τέλος δεν μπορούμε να δηλώσουμε δύο μεταβλητές με το ίδιο όνομα.

Δηλώσεις Σταθερών (Constants)

Μία σταθερά είναι κάτι το οποίο έχει ένα όνομα το οποίο “κρύβει” μία τιμή που δεν μπορεί να αλλάξει. Η χρήση των σταθερών, γίνεται για λόγους καθαρά λειτουργικούς και βοηθά στην αναγνωσιμότητα των προγραμμάτων μας. Είναι ιδιαίτερα χρήσιμες για τους προγραμματιστές, αλλά για τους αρχάριους είναι μία δύσκολη συνήθεια. Οι δηλώσεις ξεκινούν με τη λέξη **const** και έχουν τη μορφή:

ονομασία σταθεράς = τιμή ;

Π.χ.

```
const
  ProgName    = 'Designer v1.0';
  Fields      = 12;
```

Το μόνο που αξίζει κάποια προσοχή, είναι το γεγονός ότι οι σταθερές δηλώνονται μία σε κάθε γραμμή. Δεν θα ήταν άλλωστε και λογικό να δηλώνουμε δύο ή και περισσότερες σταθερές στην ίδια γραμμή. Μία χρησιμότητα ακόμα που θα μπορούσαμε να βρούμε για τις σταθερές, φαίνεται και στο προηγούμενο παράδειγμα, όπου ο προγραμματιστής δήλωσε την σταθερά ProgName που υποτίθεται ότι είναι το όνομα του προγράμματός του. Έτσι, όποτε

χρειάζεται το πρόγραμμα να τυπώνει το όνομα αυτό, στην οθόνη, στον εκτυπωτή κ.λ.π. ο προγραμματιστής θα δίνει εντολή να τυπωθεί το περιεχόμενο του ProgName και όχι το 'Designer v1.0'. Ετσι, αν σε κάποια φάση αποφασίσει να αλλάξει το 'Designer v1.0' σε 'Designer v1.1' δεν θα χρειαστεί να ψάξει σε ΟΛΟΚΛΗΡΟ το πρόγραμμα του για να βρεί πού τυπωνόταν το μήνυμα αυτό και να το διορθώσει, αλλά θα αλλάξει ΜΟΝΟ το περιεχόμενο της σταθεράς ProgName. Αυτό γλυτώνει και κόπο και χρόνο.

Constant Expressions

Μεταξύ των σταθερών η Turbo Pascal επιτρέπει να γίνονται ακόμα και πράξεις (**constant expressions**). Ετσι το παρακάτω παράδειγμα είναι σωστό :

```
const
  PlayersPerTeam = 12;
  Teams          = 4;
  AllPlayers     = PlayersPerTeam * Teams;
```

Οπως είναι φυσικό η τιμή της σταθεράς AllPlayers είναι 48. Πρέπει να προσέξουμε ότι η πράξη γίνεται από τον compiler, **πριν** από την εκτέλεση του προγράμματος! Απλά με αυτόν τον τρόπο, αν π.χ. θέλουμε να αλλάξουμε τον αριθμό των ομάδων, τότε απλά θα αλλάξουμε την τιμή της σταθεράς Teams. Η τιμή της σταθεράς AllPlayers θα γίνει αμέσως η σωστή, χωρίς να χρειαστεί να υπολογίσουμε εμείς τη νέα. Επειδή ο compiler πρέπει να μπορεί να υπολογίσει την τιμή της κάθε σταθεράς κατά τη διάρκεια του compilation απαγορεύεται να χρησιμοποιούμε στις δηλώσεις αυτές ονόματα μεταβλητών, ονόματα typed constants (βλ. παρακάτω) και κλήσεις σε συναρτήσεις εκτός από τις :

Abs, Chr, Hi, Length, Lo, Odd, Ord, Pred, Ptr, Round, SizeOf, Succ, Swap, Trunc.

Δηλώσεις Typed Constants

Οι Typed Constants (σταθερές με τύπο δεδομένων) είναι ουσιαστικά μεταβλητές. Δεν δικαιολογούν πουθενά δηλαδή την έννοια constant. Απλά έχουν το όνομα τους αυτό διότι δηλώνονται και αυτές στο κομμάτι **const** μαζί με τις απλές σταθερές που είδαμε πιο πάνω. Οι δηλώσεις τους έχουν την εξής μορφή:

όνομα typed constant : τύπος δεδομένων = αρχική τιμή ;

Π.χ.

```
const
  x          : integer = 12;
  Operators  : array[1..4] of char = ('+', '-', '*', '/');
  DaysOfMonth : array[1..12] of byte =
    (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
  s          : string[40] = '';
```

Στο παραπάνω παράδειγμα, το x είναι μία μεταβλητή τύπου integer, η οποία όμως με το που θα ξεκινήσει το πρόγραμμα, θα έχει την τιμή 12. Ιδιαίτερα βολεύουν οι typed constants για να δώσουμε τιμές στα στοιχεία κάποιων πινάκων που πολλές φορές χρησιμοποιούνται χωρίς να αλλάξουν στοιχεία (σε μενού π.χ.). Όπως φαίνεται από το παραπάνω παράδειγμα, είναι πραγματικά πολύ σύντομο να γράψουμε τις αρχικές τιμές ενός πίνακα στις δηλώσεις των typed constants.

Προσοχή. Επειδή μπορούμε να δηλώσουμε typed constants και μέσα σε procedures ή functions πρέπει να σημειωθεί ότι η τιμή τους θα είναι ίση με τη δηλωμένη ΜΟΝΟ στην αρχή του προγράμματος και όχι κάθε φορά που μπαίνουμε στην procedure ή function που έχουμε κάνει τη δήλωση.

Data Types (Τύποι Δεδομένων)

Όπως είναι γνωστό, στους υπολογιστές "μπαίνουν" και "βγαίνουν" διάφορες πληροφορίες τις οποίες συνήθως τις αποκαλούμε δεδομένα. Η μορφή των πληροφοριών αυτών δεν είναι πάντα η ίδια. Ούτε ως προς το τι δηλώνουν τα δεδομένα, ούτε ως προς το μέγεθός τους. Διαφέρουν λοιπόν το όνομα ενός πελάτη, από το υπόλοιπο ενός λογαριασμού, επειδή το ένα είναι γράμματα ενώ το δεύτερο είναι αριθμός. Επίσης διαφέρουν ο μισθός ενός εργαζομένου σε δραχμές από την ηλικία του σε χρόνια. Το ένα είναι ένας μεγάλος αριθμός (πολλά ψηφία) ενώ το άλλο είναι ένας μικρός αριθμός.

Ο τρόπος με τον οποίο καταχωρούνται τα δεδομένα στη μνήμη, οι εργασίες που μπορούμε να κάνουμε με αυτά (πράξεις) και οι τιμές που μπορούν να τους αποδωθούν λέγονται με μία ονομασία **Τύπος Δεδομένων**. Ο τ.δ. είναι κάτι που αναφέρεται σε μία πληροφορία. Λέμε δηλαδή ότι το όνομα ενός πελάτη είναι τύπου string. Είναι ευθύνη του προγραμματιστή να επιλέγει πάντα το καλύτερο δυνατό τρόπο αποθήκευσης των δεδομένων στα προγράμματά του. Οι γλώσσες προγραμματισμού δίνουν τη δυνατότητα επιλογής διάφορων τύπων δεδομένων. Ανάλογα με την περίπτωση λοιπόν μπορούμε να επιλέξουμε τον πιο κατάλληλο.

Στην Pascal, όταν ορίσουμε μια μεταβλητή, πρέπει να δηλώσουμε και τον τύπο της (τ.δ.). Ο τύπος μιας μεταβλητής όπως είπαμε πιο πάνω θα δηλώνει το σύνολο των τιμών που μπορεί να πάρει, καθώς και το ποιές πράξεις είναι δυνατό να κάνουμε χρησιμοποιώντας τη συγκεκριμένη μεταβλητή. Στη Turbo Pascal οι τύποι δεδομένων χωρίζονται σε πέντε βασικές κατηγορίες :

- **simple types** (απλοί)
- **string types** (αλφαριθμητικοί)
- **structured types** (δομημένοι)
- **pointer types** (δυναμικοί)
- **procedure types** (διαδικαστικοί)

Simple Types

Τα Simple types (απλοί τύποι δεδομένων) χωρίζονται σε δύο κατηγορίες :

- **ordinal types** (Διατεταγμένοι - Βαθμωτοί)
- **real types** (Πραγματικοί (δεκαδικοί) αριθμοί)

Simple: Ordinal Types

Οι τιμές των Ordinal Types (διατεταγμένων τύπων δεδομένων) βρίσκονται πάντα σε μια σειρά. Υπάρχει δηλαδή η πρώτη τιμή, η δεύτερη κλπ. Για κάθε τιμή ενός τύπου υπάρχει ένας αριθμός που δηλώνει σε ποιά σειρά βρίσκεται η συγκεκριμένη τιμή. Για παράδειγμα η πρώτη τιμή του τύπου **char** που είναι χαρακτήρες του κώδικα ASCII είναι ο χαρακτήρας 0 (συμβολισμός #0). Για κάθε τιμή εκτός από την τελευταία κάθε σειράς υπάρχει η επόμενη, και για κάθε τιμή εκτός από την πρώτη κάθε σειράς υπάρχει η προηγούμενη. Έτσι ο επόμενος χαρακτήρας από το A είναι ο B, και ο προηγούμενος από τον @ είναι ο !.

Simple: Ordinal: Integer Types

Οι ακέραιοι τύποι δεδομένων είναι ένα υποσύνολο των διατεταγμένων και ουσιαστικά δεν είναι τίποτε άλλο παρά απλοί αριθμοί. Υπάρχουν πέντε είδη τέτοιων τύπων και είναι οι εξής :

shortint, integer, longint, byte και word

Παρακάτω υπάρχουν παραδείγματα ορισμών τέτοιων μεταβλητών :

```
var
  TheByte      : byte;
  Int1, Int2    : integer;
  FirstNumber,
  SecondNumber : word;
```

Οι τιμές που παίρνει κάθε μεταβλητή είναι συγκεκριμένες και δεν μπορούμε να υπερβούμε τα όρια της. Υπάρχουν ορισμένοι βασικοί κανόνες που αναφέρονται στη χρήση τέτοιων μεταβλητών και είναι οι ακόλουθοι :

- Ο τύπος μιας ακέραιας σταθεράς είναι ο μικρότερος τύπος που περιλαμβάνει την τιμή της σταθεράς. Έτσι όταν η Pascal συναντήσει κάπου τον αριθμό 200 θα θεωρήσει ότι είναι byte ενώ αν συναντήσει το 40000 ότι είναι word.
- Σε μια πράξη όπου λαμβάνουν μέρος δύο αριθμοί διαφορετικών τύπων δεδομένων (π.χ. ο ένας integer και ο άλλος byte) τότε το αποτέλεσμα αυτής της πράξης είναι του τύπου εκείνου που θεωρείται κοινός και για τις δύο μεταβλητές δηλαδή που περιλαμβάνει τις τιμές και των δύο τύπων. Για παράδειγμα αν προσθέσουμε μια μεταβλητή τύπου byte με μια μεταβλητή τύπου integer το αποτέλεσμα θα είναι

integer, ενώ αν κάνουμε μια πράξη με integer και word το αποτέλεσμα θα είναι longint! (προσέξτε τους πίνακες με τις τιμές των δύο τύπων μεταβλητών και θα δείτε το γιατί).

- Η έκφραση (παράσταση) στα δεξιά ενός assignment statement υπολογίζεται ανεξάρτητα από το μέγεθος ή τύπο της μεταβλητής στα αριστερά.

Simple: Ordinal: Boolean Types

Οι μεταβλητές αυτές είναι λογικές μεταβλητές και η τιμές τους είναι αλήθεια (**true**) ή ψέμματα (**false**). Τις χρησιμοποιούμε πολύ συχνά στον προγραμματισμό για να δηλώσουμε αν κάτι συμβαίνει ή δεν συμβαίνει. Γενικά όπου στα προγράμματά μας θελήσουμε να ορίσουμε μια μεταβλητή που παίρνει δύο τιμές (σαν διακόπτης On-Off) τότε η μεταβλητή αυτή είναι συνήθως boolean. Δεν είναι γενικός κανόνας βέβαια αυτό, αλλά είναι κανόνας σωστού προγραμματισμού. Η χρήση των μεταβλητών boolean είναι περισσότερο κατανοητή.

Ας υποθέσουμε ότι γράφουμε ένα παιχνίδι και μέσα από κάποιες επιλογές του χρήστη, αυτός δηλώνει ότι θέλει να ακούγεται ήχος ή όχι. Οπως είναι φυσιολογικό χρειαζόμαστε μια μεταβλητή στην οποία θα καθορίσουμε το αν θα ακούγεται ήχος ή όχι. Εύκολα κάποιος μπορεί να πεί : "Θα ορίσω την μεταβλητή PlaySound η οποία θα είναι byte και θα έχει την τιμή 1 όταν θα ακούγεται ήχος και την τιμή 0 όταν δεν θα ακούγεται" Πράγματι αυτό "δουλεύει", και μάλιστα όπως παρακάτω :

```
if PlaySound = 1 then
begin
    Δώσε ηχητικό σήμα
end;
```

Ο σωστός προγραμματισμός όμως λέει ότι δεν είναι σωστό να χρησιμοποιώ αριθμούς εκεί που οι τιμές δεν είναι αριθμοί. Και στην περίπτωση μας το αν ακούγεται ήχος ή όχι είναι κάτι λογικό (αλήθεια ή ψέμματα) και όχι μηδέν ή ένα. Αρα μπορούμε να ορίσουμε τη μεταβλητή SoundOn η οποία είναι boolean και τότε το παραπάνω σημείο του προγράμματος θα γινόταν έτσι :

```
if SoundOn then
begin
    Δώσε ηχητικό σήμα ....
end;
```

Οι διαφορές όντως δεν είναι πολύ σημαντικές όμως σκεφτείτε ότι ο δεύτερος τρόπος είναι πιο φυσιολογικός, γιατί αν υποθέσουμε ότι κάποιος σας ρωτήσει "Ακούς τίποτα;" εσείς θα απαντήσετε με "Ναι" ή "Όχι" και όχι με "0" ή "1" !!!!

Επιπλέον δείτε το παρακάτω παράδειγμα για να καταλάβετε πόσο πιο "όμορφος" είναι ο προγραμματισμός με `boolean` μεταβλητές. Διαβάζω λοιπόν από το πληκτρολόγιο έναν χαρακτήρα και αν αυτός είναι 'Y' τότε θεωρώ ότι ο χρήστης θέλει ήχο ενώ σε κάθε άλλη περίπτωση δεν θέλει.

ΠΑΡΑΔΕΙΓΜΑ με `byte`

```
var
  Answer      : char;
  PlaySound   : byte;

begin
  write ('Sound (Y/N):');
  readln (Answer);
  if Answer = 'Y' then
    PlaySound := 1
  else
    PlaySound := 0;
  ....
  if PlaySound = 1 then
    begin
      ..... παίζει ήχους
    end;
  ....
end.
```

ΠΑΡΑΔΕΙΓΜΑ με `boolean`

```
var
  Answer      : char;
  PlaySound   : boolean;

begin
  write ('Sound (Y/N):');
  readln (Answer);
  PlaySound := (Answer='Y');
  ....
  if PlaySound then
    begin
      ..... παίζει ήχους
    end;
  ....
end.
```

Στο παράδειγμα που βρίσκεται στα δεξιά η πρόταση
`PlaySound := (Answer = 'Y');`

σημαίνει ότι η μεταβλητή `SoundOn` θα πάρει σαν τιμή το αποτέλεσμα της λογικής πρότασης `(Answer = 'Y')` το οποίο είναι `true` αν η μεταβλητή `Answer` είναι 'Y' και `false` όταν δεν είναι.

Simple: Ordinal: Char Type

Οι μεταβλητές τύπου `char` είναι απλοί χαρακτήρες του κώδικα ASCII. Χρησιμοποιούνται για αποθήκευση **ενός μόνο** χαρακτήρα. Οι χαρακτήρες στην Pascal όπως είναι γνωστό γράφονται μέσα σε μονά εισαγωγικά (απόστροφος). Χρειάζεται προσοχή το εξής : Όταν μέσα στα εισαγωγικά υπάρχει **ΜΟΝΟ ένας** χαρακτήρας τότε η Pascal το θεωρεί αυτό **char**, αλλιώς το θεωρεί **string**. Για να καταχωρήσουμε λοιπόν σε μια μεταβλητή αυτού του τύπου έναν χαρακτήρα απλά τον γράφουμε μέσα σε εισαγωγικά. Σε περίπτωση που ξέρουμε τον κωδικό ASCII του χαρακτήρα, μπορούμε να χρησιμοποιήσουμε τον συμβολισμό `#nnn` όπου `nnn` ένας αριθμός από 0 έως 255.

Ετσι είναι το ίδιο αν πούμε :

```
AChar := #64;
AChar := '@';
```

Simple: Ordinal: Enumerated Types

Τα enumerated types (απαριθμητοί τύποι) είναι τύποι δεδομένων που τους γράφουμε εξ' ολοκλήρου μόνοι μας. Χρησιμεύουν συνήθως για να κάνουν το πρόγραμμά μας πιο ευανάγνωστο και για να ξεφύγουμε από κωδικοποίηση τιμών με νούμερα. Αν θέλουμε να έχουμε μία μεταβλητή που να κρατά το είδος του χρώματος ενός χαρτιού της τράπουλας (κούπα, καρρώ, σπαθί και μπαστούνι), μπορούμε να τη θέσουμε ως ένα byte και να χρησιμοποιούμε τις τιμές 0..3 ή 1..4. Τα προβλήματα που θα δημιουργηθούν είναι βέβαια πολλά. Θα φτάσουμε στο σημείο να γράφουμε :

```
CardColor := 2;
if CardColor = 3 then
    ....
for CardColor := 0 to 3 do
    ....
```

Ολα τα παραπάνω θα δουλέψουν, θα κάνουμε τη δουλειά μας, αλλά η ουσία είναι μετά από λίγο καιρό, αν κοιτάξουμε στο πρόγραμμά μας, τι θα καταλάβουμε όταν δούμε ότι το CardColor το κάνουμε ίσο με 2; Θα μπορούσαμε να βάλουμε comments αλλά το νόημα στην Pascal δεν είναι να γράφουμε comments για να καταλαβαίνουμε τον κώδικά μας, αλλά να γράφουμε κώδικα που "μιλάει μόνος του" (self-documenting code).

Τα enumerated types μας δίνουν ακριβώς αυτό. Μας δίνουν τη δυνατότητα να περιγράψουμε με λέξεις τις τιμές ενός τύπου δεδομένων καινούργιου, ενός τύπου δεδομένων εντελώς δικού μας. Ετσι λοιπόν δηλώνουμε :

```
type
    TCardColor = (Heart, Spade, Diamond, Club);
```

Η παραπάνω δήλωση λέει ότι ο τύπος δεδομένων TCardColor παίρνει τις τιμές Heart, Spade, Diamond και Club. Αυτές και μόνο αυτές. Ακόμα, οι τιμές αυτές είναι διατεταγμένες, δηλαδή πρώτη τιμή είναι το Heart και τελευταία το Club. Μην ξεχνάτε ποτέ ότι τα enumerated types είναι βασικά Ordinals άρα μπορούν να χρησιμοποιηθούν σε όλες τις προτάσεις (for, case) αλλά και στις δηλώσεις πινάκων, όπως θα δούμε.

Το πρόγραμμά μας με το TCardColor μπορεί πλέον να γίνει πολύ ευανάγνωστο. Σε αντίθεση με τα προηγούμενα παραδείγματα που γράψαμε όταν χρησιμοποιούσαμε κωδικοποίηση με αριθμούς τύπου byte, οι προτάσεις μπορούν να γραφτούν ως εξής :

```
var
    CardColor : TCardColor;
begin
    CardColor := Diamond;
    if CardColor = Spade then
        ....
```

```

    for CardColor := Heart to Club do
        ....
    end.

```

Ενα λάθος που κάνουν όλοι όσοι πρωτογνωρίζουν τα enumerated types είναι να νομίζουν ότι είναι σαν strings. Οι λέξεις που χρησιμοποιούμε δεν είναι τίποτε άλλο από κάτι σαν σύμβολα. Στην ουσία η Pascal τα μεταφράζει σαν νούμερα. Απλώς εμείς τα χειριζόμαστε σαν λέξεις. Εκεί είναι που το πρόγραμμά μας γίνεται περισσότερο ευανάγνωστο.

Είναι λάθος λοιπόν να πούμε :

```

writeln (CardColor);

```

γιατί απλούστατα η Pascal δεν ξέρει πώς να τυπώσει μία μεταβλητή τύπου TCardColor. Ξέρει να τυπώνει strings, ξέρει να τυπώνει νούμερα, αλλά όχι αυτό το καινούργιο - user defined TCardColor. Στο συγκεκριμένο παράδειγμα των "χρωμάτων" της τράπουλας, μπορούμε να φτιάξουμε έναν μικρό πίνακα που να έχει τις περιγραφές των χρωμάτων (ακόμα και στα ελληνικά) και να τυπώνουμε την αντίστοιχη περιγραφή, για την αντίστοιχη τιμή του TCardColor :

```

const
    Description : array[Heart..Club] of string[10] =
        ('Κούπα', 'Μπαστούνι', 'Καρώ', 'Σπαθί');

```

Με τα enumerated types στην Pascal καλύπτουμε πλέον κάθε περίπτωση δεδομένων. Ο,τι μπορούμε να σκεφτούμε σαν data, μπορούμε να βρούμε και τον κατάλληλο τύπο για να το κρατήσουμε στη μνήμη. Η δύναμη της Pascal είναι ακριβώς εδώ. Πολλοί έτοιμοι τύποι (byte, integer, boolean, reals κ.λ.π.) αλλά και τύποι που μπορούμε να φτιάξουμε μόνοι μας με όλη τη λειτουργικότητα που μπορούμε να φανταστούμε.

Simple: Real Types

Οι τύποι αυτοί ανήκουν στους απλούς τύπους της Pascal αλλά όχι και στους ordinal. Είναι ουσιαστικά η δεύτερη υποκατηγορία των απλών. Παριστάνουν τους πραγματικούς αριθμούς και τα είδη τους είναι πέντε :

real, single, double, extended και comp

Τα τέσσερα τελευταία είδη απαιτούν ο υπολογιστής μας να έχει μαθηματικό συνεπεξεργαστή (πέρα από την κεντρική μονάδα επεξεργασίας) αλλά μπορούμε με κάποιο switch της Pascal να δηλώσουμε ότι κάνουμε προσομοίωση αυτού του επεξεργαστή χωρίς ουσιαστικά να υπάρχει στον υπολογιστή μας. Επειδή όμως έτσι κι αλλιώς ο τύπος **real** μας καλύπτει, η χρήση όλων των υπόλοιπων τύπων είναι σπάνια (μόνο σε εξαιρετικά απαιτητικές επιστημονικές εφαρμογές που χρειαζόμαστε πολύ μεγάλη ακρίβεια). Με τις μεταβλητές τύπου **real** μπορούμε να κάνουμε όλες τις γνωστές αριθμητικές πράξεις.

String Types

Οι μεταβλητές αυτές είναι μια ξεχωριστή κατηγορία μεταβλητών (αλφαριθμητικές). Αποτελούνται από μια σειρά χαρακτήρων και έναν αριθμό που δηλώνει το μέγεθός τους. Έτσι στη μνήμη ένα string 40 χαρακτήρων κρατάει 41 bytes. 40 για τους χαρακτήρες και 1 για το μέγεθός του. Επειδή δε σε ένα byte μπορούμε να βάλουμε έναν αριθμό από 0 έως 255 αντιλαμβανόμαστε ότι το μέγεθος ενός string είναι 0 έως 255 χαρακτήρες.

Όταν δηλώνουμε μεταβλητές τύπου string στην Pascal πρέπει να γράψουμε δίπλα από τη λέξη, πόσους χαρακτήρες το πολύ θα μπορεί να κρατήσει η μεταβλητή μας. Έτσι αν πούμε λοιπόν ότι η μεταβλητή *s* είναι `string[20]` σημαίνει ότι ανά πάσα στιγμή στο *s* θα υπάρχουν από 0 μέχρι 20 χαρακτήρες. Μηδέν (0) χαρακτήρες σημαίνει ότι το *s* είναι κενό (άδειο - empty string) και δεν έχει χαρακτήρες. Αυτό συμβολίζεται ανοίγοντας και κλείνοντας αμέσως εισαγωγικά. Π.χ. `s := ''`;

Υπάρχουν δύο τρόποι να δηλώσουμε ότι μια μεταβλητή είναι string και είναι οι εξής:

```
var
  OneString   : string[40]; { max μέγεθος 40 chars }
  OtherString : string;     { max μέγεθος 255 chars }
```

Καταλαβαίνουμε βέβαια ότι string και string[255] είναι το ίδιο πράγμα!

Παραδείγματα :

- `OneString := 'something';`
Δίνω στο OneString την τιμή 'something'.
- `OtherString := OneString+'*'+OneString;`

Κάνω πρόσθεση strings, δηλαδή συνένωση τους.

- `OneString := '';`

Δίνω στο `OneString` την τιμή `"` που είναι το κενό string (μέγεθος 0). Προσέξτε ότι γράφουμε δύο φορές την απλή απόστροφο (`'`) και όχι το διπλό εισαγωγικό (`"`).

Σχέση String και Char

Οι τύποι `string` και `char` έχουν κάποια σχέση, αλλά υπάρχουν κάποιοι περιορισμοί στην χρήση τους. Στα strings ισχύει η πράξη της πρόσθεσης και έχει σαν αποτέλεσμα να ενώσει δύο strings. Στα chars αυτό δεν ισχύει. Όπως ήδη έχουμε πεί, αν η Pascal δει έναν χαρακτήρα ανάμεσα σε δύο εισαγωγικά θα το θεωρήσει ένα `char`, ενώ άμα δει δύο ή περισσότερους ή και κανέναν (!) θα το θεωρήσει `string`. Ισχύουν τα παρακάτω (`s : string` και `c : char` παντού):

<code>string <- char</code>	π.χ. <code>s := c;</code> ή <code>s := '@';</code>
<code>char+string => string</code>	η πρόσθεση <code>string</code> με <code>char</code> μας δίνει <code>string</code>
<code>char+char => string</code>	η πρόσθεση δύο <code>chars</code> μας δίνει και πάλι <code>string</code> .

Type Compatibility (Συμβατότητα Τύπων Δεδομένων)

Η συμβατότητα μεταξύ δύο τύπων δεδομένων είναι απαιτούμενη ιδιαίτερα σε εκφράσεις ή σχέσεις μεταξύ μεταβλητών. Ακόμα η συμβατότητα τύπων δεδομένων είναι και προϋπόθεση συμβατότητας για assignment statements. Αυτό που ονομάζουμε `assignment compatibility` είναι απαραίτητη όταν αντικαθιστούμε με μια τιμή μια μεταβλητή. Ένας τύπος `T2` είναι `assignment compatible` με έναν τύπο `T1` (σημαίνει ότι μπορώ να γράψω `T1 := T2`) αν συμβαίνει ένα από τα παρακάτω :

- `T1` και `T2` είναι ίδιοι τύποι (αλλά όχι `file types` = αρχεία)
- `T1` και `T2` είναι συμβατοί `ordinal` τύποι και οι τιμές του `T2` ανήκουν μέσα στο σύνολο των τιμών του `T1`.
- `T1` και `T2` είναι `real types` και οι τιμές του `T2` ανήκουν στο σύνολο των τιμών του `T2`.
- `T1` είναι `real` και `T2` είναι `integer`.
- `T1` και `T2` είναι `strings`.
- `T1` είναι `string` και `T2` είναι `char`.

Expressions και Operators

Μια έκφραση κάποιου τύπου δεδομένων είναι μία σταθερά, μία μεταβλητή το αποτέλεσμα μίας συνάρτησης ή όλα τα προηγούμενα συνδυασμένα σε πράξεις που έχουν σαν τελικό αποτέλεσμα μία τιμή του τύπου αυτού. Οτιδήποτε δηλαδή “βγάζει” μία τιμή κάποιου τύπου δεδομένων θεωρείται μία έκφραση. Σε πάρα πολλά σημεία της σύνταξης της γλώσσας, λέμε ότι π.χ. χρειάζεται μία boolean έκφραση. Αυτό σημαίνει: κάτι που να μας δίνει boolean.

Για τις περιπτώσεις που βάζω μία σταθερά ή έστω το όνομα μιας μεταβλητής έχει καλώς. Τα σημεία που θα εξετάσουμε στο Κεφάλαιο αυτό, είναι οι πράξεις που γίνονται μεταξύ των στοιχείων κάποιων τύπων και τα αποτελέσματα που βγάζουν. Οι πράξεις τελούνται με κάποια σύμβολα (operators - τελεστές). Πολλά απ' αυτά θα συζητηθούν και στα κεφάλαια που ασχολούνται με συγκεκριμένους τύπους δεδομένων (π.χ. strings, sets).

Operators (Τελεστές)

Οι τελεστές χωρίζονται σε διάφορες κατηγορίες ανάλογα με τις πράξεις τις οποίες τελούν. Είναι οι τελεστές αριθμητικών πράξεων, οι τελεστές λογικών πράξεων μεταξύ boolean τιμών, ένας τελεστής για strings, τελεστές για σύνολα (sets) και τελεστές σύγκρισης (κυρίως για αριθμούς, αλλά και για char και strings). Υπάρχει περίπτωση κάποιοι τελεστές να είναι κοινοί σε διαφορετικές κατηγορίες. Αυτό δεν σημαίνει ότι απαραίτητα υπάρχει όμοια σχέση. Π.χ. το “συν” (+) που στους αριθμούς κάνει πρόσθεση, υπάρχει και στα strings όπου κάνει συνένωση, αλλά ακόμα και στα σύνολα που κάνει **ένωση**. Παρακάτω υπάρχουν συνοπτικοί πίνακες με όλους τους τελεστές της Turbo Pascal.

Arithmetic Operators (Αριθμητικοί Τελεστές)

Τελεστής	Πράξη	Παίρνουν μέρος στην πράξη	Τύπος Αποτελέσματος
+	Πρόσθεση	integers reals	integer real

-	Αφαίρεση	integers reals	integer real
*	Πολλαπλασιασμός	integers reals	integer real
/	Διαίρεση	integers, reals	real
div	Ακέραια Διαίρεση	integers	integer
mod	Υπόλοιπο Διάρεσης	integers	integer

Boolean Operators (Λογικοί Τελεστές)

Τελεστής	Πράξη	Παίρνουν μέρος	Αποτέλεσμα
not	άρνηση	ένα boolean	boolean
and	σύζευξη	booleans	boolean
or	διάζευξη	booleans	boolean
xor	αποκλειστική διάζευξη (ή το ένα ή το άλλο)	booleans	boolean

String Operator

Στα strings ισχύει μόνο η πρόσθεση (+) που έχει σαν αποτέλεσμα την ένωση δύο strings σε ένα. Επίσης πρόσθεση μπορούμε να κάνουμε και σε δύο chars με αποτέλεσμα ένα string!!!

Set Operators (Τελεστές Συνόλων)

Τελεστής	Πράξη	Παίρνουν μέρος και δίνουν αποτέλεσμα
+	Ένωση	Συμβατά sets
-	Διαφορά	Συμβατά sets
*	Τομή	Συμβατά sets

Relational Operators (Τελεστές Σύγκρισης)

Οι τελεστές σύγκρισης συγκρίνουν δύο τιμές συμβατών τύπων δεδομένων και δίνουν σαν αποτέλεσμα boolean (true ή false). Στην περίπτωση των strings, ένα string θεωρείται μικρότερο από ένα άλλο, όταν είναι κατώτερο σε **αλφαβητική σειρά** (σύμφωνα πάντα με

τον πίνακα ASCII). Δηλαδή το 'ARIS' είναι μικρότερο από το 'BOO' ανεξαρτήτως πλήθους γραμμάτων όπως πολλοί πιστεύουν (λανθασμένα).

Τελεστής	Πράξη	Παίρνουν μέρος
=	ισότητα	συμβατοί απλοί, sets, strings
<>	διάφορο	συμβατοί απλοί, sets, strings
<	μικρότερο	συμβατοί απλοί, strings
>	μεγαλύτερο	συμβατοί απλοί, strings
<=	μικρότερο ή ίσο	συμβατοί απλοί, strings
>=	μεγαλύτερο ή ίσο	συμβατοί απλοί, strings
<=	υποσύνολο	συμβατά sets
>=	υπερσύνολο	συμβατά sets
in	ανήκει	αριστερά: κάποιος ordinal τύπος δεξιά: κάποιο set αυτού του τύπου

ΚΕΦΑΛΑΙΟ 5

Statements (Προτάσεις)

Οι προτάσεις στην Turbo Pascal μπορούν να χωριστούν κατά τον ακόλουθο τρόπο :

- **Simple** (Απλές)
 - **Assignment** (Αντικατάστασης)
 - **Procedure** (Διαδικασιών)
- **Structured** (Δομημένες)
 - **Compound** (Σύνθετες)
 - **Conditional** (Υποθετικές)
 - **Repetitive** (Επαναληπτικές)
 - **With**

Simple Statements (Απλές Προτάσεις)

Μια απλή πρόταση είναι μία πρόταση που δεν περιέχει καμία άλλη.

Assignment Statements (Προτάσεις Αντικατάστασης)

Αυτές αντικαθιστούν το περιεχόμενο μιας μεταβλητής με μια νέα τιμή, σύμφωνα με την ακόλουθη σύνταξη :

μεταβλητή := έκφραση του ίδιου τύπου δεδομένων με τη μεταβλητή ;

Έκφραση είναι οποιαδήποτε σταθερά, μεταβλητή, αποτέλεσμα μιας συνάρτησης ή αποτέλεσμα πράξης με όλα τα προηγούμενα.

Παραδείγματα:

```
X := Y+Z;  
I := Sqr(J)-I*K;
```

Procedure Statements (Κλήση Διαδικασιών)

Οι προτάσεις αυτές δηλώνουν την ενεργοποίηση μιας διαδικασίας. Γράφουμε το όνομα της διαδικασίας και τις παραμέτρους που χρειάζεται για να πραγματοποιηθεί.

Παραδείγματα:

```
PrintHeading;
```

```
Transpose (A,N,M);  
Find ('John',Address);
```

Structured Statements (Δομημένες Προτάσεις)

Οι δομημένες προτάσεις είναι αποτελέσματα σύνθεσης άλλων προτάσεων που θα εκτελεστούν με τη σειρά (with και begin-end), υποθετικά (if, case) ή επαναληπτικά.

Compound Statements (Σύνθετες Προτάσεις)

Σε μία σύνθετη πρόταση, οι υπο-προτάσεις που την αποτελούν εκτελούνται η μία μετά την άλλη με τη σειρά που έχουν γραφτεί. Η χρησιμότητα τους είναι, ότι κατά κάποιον τρόπο “δένουν” περισσότερες από μία προτάσεις σε ένα block που καθορίζεται από ένα begin στην αρχή και ένα end στο τέλος και όλο αυτό το group προτάσεων θεωρείται μετά από την Pascal σαν μία ολοκληρωμένη πρόταση.

Παράδειγμα:

```
begin  
  writeln ('Enter a number : ');  
  readln (x);  
  writeln ('Thanks! The number was ',x);  
end;
```

Conditional Statements (Υποθετικές Προτάσεις)

Μία υποθετική πρόταση επιλέγει για εκτέλεση μία ή και καμία από τις υπο-προτάσεις της. Οι υποθετικές προτάσεις είναι δύο. Η if - then και η case - of. Μία παραλλαγή θα λέγαμε της if - then είναι και η if - then - else που συνήθως αναφέρεται και σαν τρίτη υποθετική πρόταση.

- **if - then**

if boolean έκφραση then

πρόταση;

Σε περίπτωση που η boolean έκφραση είναι **true** τότε εκτελείται η πρόταση, αλλιώς δεν γίνεται τίποτα. Προσέξτε ότι η πρόταση θα μπορούσε να είναι οποιαδήποτε πρόταση της Pascal. Είτε απλή, είτε σύνθετη (ένα begin - end που περιέχει πολλές άλλες), είτε μία άλλη if, είτε μία for κ.λ.π. Π.χ.

```
if c = 'Q' then  
begin  
  writeln ('Exiting game...');  
  QuitGame;  
end;
```

- **if - then - else**

if boolean έκφραση then

πρόταση1

else

πρόταση2;

Όταν η boolean έκφραση είναι **true** τότε εκτελείται η πρόταση 1 αλλιώς εκτελείται η πρόταση 2. Π.χ.

```
if a = 1 then
  writeln ('*')
else
  Stars (a);
```

- **case - of**

Η case αποτελείται από μία έκφραση (τον επιλογέα) και μία λίστα από προτάσεις που έχουν στην αρχή τους κάποιες σταθερές (επιλογής) ή τη λέξη else. Ο επιλογέας πρέπει να είναι ordinal, με μέγεθος byte ή word. Αρα, οι τύποι longint, string κ.α. δεν είναι δεκτοί. Όλες οι σταθερές επιλογής πρέπει να είναι μοναδικές και του ίδιου (ή συμβατού) τύπου με την ordinal έκφραση (επιλογέα).

case ordinal expression **of**

constant : πρόταση ;

constant..constant : πρόταση ;

constant,... : πρόταση ;

else : πρόταση ;

end;

Παράδειγμα:

```
case A of
1      : writeln ('one');
2,3    : writeln ('two or three');
5..10  : writeln ('five to ten');
else   : writeln ('something different');
end;
```


Repetitive Statements (Επαναληπτικές Προτάσεις)

Οι επαναληπτικές προτάσεις έχουν σα σκοπό να επαναλάβουν πολλές φορές την εκτέλεση κάποιας ή κάποιων άλλων προτάσεων. Έχουμε τρία είδη επαναληπτικών προτάσεων που καλύπτουν πλήρως όλες τις ανάγκες για επανάληψη. Είναι η **for - do** που χρησιμοποιείται όταν ξέρουμε εκ των προτέρων τον αριθμό των επαναλήψεων που πρέπει να γίνουν, η **repeat - until** και η **while - do** που χρησιμοποιούνται σε κάθε άλλη περίπτωση.

Η λογική για την επιλογή μιας από τις τρεις προτάσεις βασίζεται στο πόσες επαναλήψεις θέλουμε να κάνουμε. Αν ξέρουμε τον αριθμό τους επιλέγουμε την **for - do**. Αν δε γνωρίζουμε πόσες επαναλήψεις πρέπει να γίνουν, τότε σε περίπτωση που πρέπει οι προτάσεις να εκτελεστούν ΤΟΥΛΑΧΙΣΤΟΝ μία φορά, χρησιμοποιούμε την **repeat - until**. Αν όμως οι προτάσεις μπορούν και να μην εκτελεστούν και καθόλου, χρησιμοποιούμε τη **while - do**.

- **for - do**

Η πρόταση **for** εκτελεί μία πρόταση πολλές φορές, στο διάστημα που μία μεταβλητή (μετρητής) πάρει τιμές, από μία αρχική τιμή μέχρι μία τελική. Με τον τρόπο αυτό γνωρίζουμε τον αριθμό των επαναλήψεων. Η πρόταση που θα εκτελεστεί μπορεί κάλλιστα να είναι μία σύνθετη πρόταση (**begin - end**).

for ordinal τοπική μεταβλητή := έκφραση **to** έκφραση **do**

πρόταση ;

ή

for ordinal τοπική μεταβλητή := έκφραση **downto** έκφραση **do**

πρόταση ;

Μόλις αρχίζει να εκτελείται η **for** η αρχική και η τελική τιμή υπολογίζονται από τις δύο εκφράσεις (σε περίπτωση που δεν είναι σταθερές αλλά π.χ. αποτελέσματα πράξεων) και δεν μπορούν να αλλάξουν στη διάρκεια. Η εκτέλεση θα αρχίσει και θα συνεχιστεί μέχρι η μεταβλητή μας να πάρει την τελική τιμή. Σε περίπτωση που θέλουμε η μεταβλητή να αυξάνει χρησιμοποιούμε το **to**, ενώ αν θέλουμε η μεταβλητή να μειώνεται χρησιμοποιούμε το **downto**. Αν (στην περίπτωση **to**) η αρχική τιμή είναι μεγαλύτερη από την τελική, τότε η υποπρόταση δεν εκτελείται καθόλου! Το ίδιο ισχύει και στην περίπτωση **downto**, όταν η αρχική τιμή είναι μικρότερη από την τελική.

Παράδειγμα:

```
for f := 1 to 10 do  
  writeln (f);
```

- **repeat - until**

Η repeat - until επαναλαμβάνει ένα group προτάσεων (η σύνταξή της επιτρέπει πολλές προτάσεις χωρίς την ανάγκη χρήσης begin - end (σύνθετης)) μέχρι να ισχύσει μία συνθήκη. Η repeat - until θα εκτελέσει **οπωσδήποτε τουλάχιστον μία φορά** τις προτάσεις.

```
repeat
    πρόταση;
    ....
until boolean expression;
```

Οι προτάσεις που περιέχονται στο repeat - until θα εκτελούνται μέχρι η boolean έκφραση να μας δώσει αποτέλεσμα **true**. Παράδειγμα:

```
repeat
    write ('Enter value (0..9) : ');
    readln (a);
until (a >= 0) and (a <= 9);
```

- **while - do**

Η while εκτελεί μία πρόταση για όσο διάστημα διαρκεί μία συνθήκη. Αν όμως εξ αρχής η συνθήκη δεν ισχύει τότε η πρόταση δεν εκτελείται ποτέ!

```
while boolean expression do
    πρόταση ;
```

Παράδειγμα:

```
while Data[i] <> x do
    i := i+1;
```

With Statements

Η πρόταση with είναι μια πρόταση που μας βολεύει στον χειρισμό των records. Ενώ δηλαδή τα πεδία των records για να τα χειριστούμε και π.χ. να βάλουμε μέσα τιμές, πρέπει να τα γράφουμε με το όνομα του record πρώτα και μετά το όνομα του πεδίου χωρισμένα με μία τελεία (AStudent.Name), μέσα σε μία πρόταση with το όνομα του record είναι περιττό και γράφουμε μόνο το όνομα του πεδίου. Αυτό μας γλυτώνει συνήθως από πολύ γράψιμο.

```
with μεταβλητή τύπου record do
    πρόταση ;
```

Ετσι λοιπόν το παρακάτω παράδειγμα:

```
writeln (OneStudent.Name, ' ', OneStudent.Address);
```

μπορεί να γραφτεί με τη βοήθεια της πρότασης with έτσι:

```
with OneStudent do
    writeln (Name, ' ', Address);
```

Procedures και Functions

Οι procedures και οι functions είναι τα δομικά υλικά ενός προγράμματος της Pascal. Όπως έχουμε ήδη αναφέρει το κυρίως πρόγραμμα είναι συνήθως μικρό (5-10 γραμμές). Όλη η ροή του προγράμματος γίνεται “καλώντας” κάποιες υπορουτίνες, δηλαδή τις procedures και τις functions. Οι **procedures** (**διαδικασίες**) συνήθως τελούν κάποια εργασία και τις καλούμε με μία procedural πρόταση (χρησιμοποιώντας δηλαδή το όνομα τους). Π.χ. ταξινομούν έναν πίνακα. Οι **functions** (**συναρτήσεις**) συνήθως εκτελούν κάποιες πράξεις ή άλλες εργασίες αλλά **πάντα επιστρέφουν** κάποιο αποτέλεσμα. Τις καλούμε όχι μόνες τους, αλλά πάντα σαν μέρος μίας έκφρασης όπου το αποτέλεσμα τους χρησιμοποιείται ανάλογα. Π.χ. θα μπορούσε μία function να υπολογίσει και να μας επιστρέψει τον μέσο όρο των στοιχείων κάποιου πίνακα κ.λ.π.

Procedure Declarations (Δηλώσεις Διαδικασιών)

Μια δήλωση procedure ουσιαστικά αντιστοιχεί ένα όνομα σε ένα block από προτάσεις. Αργότερα μπορούν αυτές οι προτάσεις να εκτελεστούν χρησιμοποιώντας το όνομα της procedure. Μία procedure αποτελείται από την επικεφαλίδα της, τις δηλώσεις και το κυρίως μέρος εκεί δηλαδή που περιέχονται οι προτάσεις που την αποτελούν.

```
procedure Όνομα ( παράμετροι ) ;  
    Δηλώσεις  
begin  
    Προτάσεις  
end;
```

Στην επικεφαλίδα μίας procedure γράφουμε το όνομα της και αν υπάρχουν τις παραμέτρους της μέσα σε παρενθέσεις.

Παρακάτω είναι ένα απλό παράδειγμα μίας procedure που δεν έχει παραμέτρους και τυπώνει πέντε αστεράκια (*):

```

procedure PrintFiveStars;
begin
    writeln ( '*****' );
end;

```

Σε μία `procedure` μπορούμε όμως να βάλουμε και παραμέτρους. Με τον τρόπο αυτό μπορούμε να “στείλουμε” στην `procedure` κάποιες τιμές προκειμένου να αλλάξουμε τον τρόπο με τον οποίο θα λειτουργήσει ή να της δώσουμε τη δυνατότητα να χειρίζεται κάποια δεδομένα, διαφορετικά κάθε φορά που την καλούμε. Στο προηγούμενο παράδειγμα η `procedure` που φτιάξαμε τυπώνει απλά πέντε αστεράκια κάθε φορά που την καλούμε. Αυτό θα ήταν πολύ χρήσιμο αν σε ένα πρόγραμμα η μόνη μας ανάγκη θα ήταν να τυπώνουμε πέντε αστεράκια μαζί. Πόσο πιο λειτουργικό όμως θα ήταν να είχαμε μία `procedure` που να μπορεί να τυπώνει όσα αστεράκια της πούμε, ακριβώς τη στιγμή που θα την καλέσουμε; Αυτό ακριβώς το πρόβλημα, ξεπερνιέται με τις παραμέτρους. Στο επόμενο παράδειγμα η `procedure` `PrintStars` “δέχεται” έναν αριθμό `n` και τυπώνει `n` αστεράκια. Αυτή τη στιγμή δεν ξέρουμε ποιός είναι αυτός ο αριθμός. Το μόνο που ξέρουμε είναι ότι όταν θα χρειαστεί να την καλέσουμε είμαστε υποχρεωμένοι να δώσουμε εμείς αυτόν τον αριθμό.

```

procedure PrintStars ( n : byte );
var
    i : byte;
begin
    for i := 1 to n do
        write ( '*' );
    writeln;
end;

```

Στο παραπάνω παράδειγμα θα προσέξατε και τη δήλωση μίας τοπικής μεταβλητής, της μεταβλητής `i`. Για τις τοπικές μεταβλητές θα μιλήσουμε σ' αυτό το Κεφάλαιο παρακάτω.

Function Declarations (Δηλώσεις Συναρτήσεων)

Μία function είναι ένα block κώδικα, ένα σύνολο από προτάσεις δηλαδή, που υπολογίζει και επιστρέφει μία τιμή (αποτέλεσμα). Η μορφή της είναι ακριβώς σαν αυτή μιας procedure, η μόνη τους διαφορά είναι στην επικεφαλίδα τους.

```
function Όνομα ( παράμετροι ) : τύπος δεδομένων αποτελέσματος ;  
    Δηλώσεις  
begin  
    Προτάσεις  
end;
```

Στην επικεφαλίδα μιας function γράφουμε το όνομα της, τις παραμέτρους της μέσα σε παρενθέσεις (αν υπάρχουν) και τέλος τον τύπο δεδομένων του αποτελέσματος που επιστρέφει. Παρακάτω είναι ένα παράδειγμα μιας function που επιστρέφει τον μικρότερο από δύο ακέραιους αριθμούς:

```
function Min ( a,b : integer ) : integer;  
begin  
    if a < b then  
        Min := a  
    else  
        Min := b;  
end;
```

Μια function επιστρέφει την τιμή (το αποτέλεσμα) της, όταν τελειώσουν οι προτάσεις και φτάσει στο **end**;. Η τιμή που επιστρέφει πρέπει να έχει καταχωρηθεί στο όνομα της, το οποίο μέσα στη συνάρτηση θεωρείται σαν μία “ειδική” μεταβλητή. Στο παραπάνω παράδειγμα το όνομα της συνάρτησης (Min) χρησιμοποιείται σαν μία μεταβλητή integer στην οποία καταχωρούμε αυτό που πρέπει να επιστραφεί. Αν δηλαδή ο αριθμός a είναι μικρότερος από τον b, “βάζουμε” στο Min την τιμή του a (Min := a;). Βέβαια, μπορεί να θεωρείται το όνομα μίας function μεταβλητή, αλλά αυτό ισχύει μόνο μέσα στο “σώμα” της και μπορεί να χρησιμοποιηθεί μόνο στα αριστερά του ίσον (:=) της αντικατάστασης. Αποκλειστικά μόνο δηλαδή, για να βάλουμε μία τιμή μέσα της. Το παραπάνω παράδειγμα μπορεί να γραφτεί και λίγο διαφορετικά:

```
function Min ( a,b : integer ) : integer;  
begin  
    Min := b;  
    if a < b then  
        Min := a;  
end;
```

Όπως φαίνεται και από το τελευταίο αυτό παράδειγμα μπορούμε να βάλουμε πολλές φορές μία τιμή μέσα στο Min. Η τιμή που θα επιστραφεί θα είναι η τελευταία που “μπήκε” όταν θα έχουμε φτάσει στο **end**;

Local Variables (Τοπικές Μεταβλητές)

Σε αντίθεση με τις global μεταβλητές που δηλώνονται στην αρχή ενός προγράμματος της Pascal, υπάρχουν και οι local μεταβλητές που δηλώνονται μέσα σε procedures και functions, οι οποίες χρησιμεύουν μόνο μέσα στις ρουτίνες που δηλώνονται. Οι local μεταβλητές εξυπηρετούν πάρα πολύ διότι, δεν πιάνουν χώρο στη μνήμη παρά μόνο τη στιγμή που η procedure ή η function που ανήκουν είναι ενεργοποιημένη. Αυτό γλιτώνει πάρα πολύ μνήμη και λύνει τα χέρια των προγραμματιστών. Πως όμως θα γνωρίζουμε αν κάποια μεταβλητή πρέπει να την κάνουμε global ή local; Ο κανόνας που ισχύει σ' αυτές τις περιπτώσεις είναι:

Όλες τις μεταβλητές μας προσπαθούμε να τις κάνουμε local, εκτός κι αν αυτό είναι αναπόφευκτο, επειδή χρειάζονται και για την εκτέλεση άλλων ρουτινών.

Γενικά global είναι οι μεταβλητές που χρειάζεται να τις “γνωρίζουν” σχεδόν όλες οι ρουτίνες μας (αρχεία, records, κάποιοι μεγάλοι πίνακες κ.λ.π.) γιατί όλες τις χρησιμοποιούν.

ΠΡΟΣΟΧΗ. Από την έκδοση 5.0 της Turbo Pascal οι μεταβλητές που χρησιμοποιούνται στην πρόταση for είναι υποχρεωτικά local. Αυτό ήταν μία πολύ καλή σκέψη, παρόλο που στην αρχή φαίνεται αρκετά αυστηρό από μέρους της γλώσσας, διότι πάρα πολλά προβλήματα έχουν δημιουργηθεί από τη χρήση global μεταβλητών μέσα σε for στο παρελθόν.

Ορισμός Παραμέτρων

Είναι πολύ πιθανό όπως είδαμε και παραπάνω, να θέλουμε να “στείλουμε” σε μια procedure ή function κάποιες παραμέτρους. Η σύνταξη της επικεφαλίδας και για τις procedures και για τις functions είναι η ίδια. Ανοίγουμε παρένθεση και εκεί πέρα γράφουμε τις παραμέτρους που θέλουμε να δέχεται η procedure ή η function μας, δίνοντας τους ονόματα. Υπάρχουν **τρεις** τρόποι να περάσουμε παραμέτρους στις ρουτίνες μας. Οι **value**, **constant** και

variable parameters. Παρόλο που φαινομενικά εξυπηρετούν τον ίδιο σκοπό, στην ουσία έχουν σημαντικές διαφορές.

Value Parameters

Οι παράμετροι αυτές, στέλνουν στη ρουτίνα μία τιμή κάποιου τύπου δεδομένων. Η ρουτίνα χρησιμοποιεί την τιμή αυτή σε ότι της χρειάζεται. **Η παράμετρος ενεργεί μέσα στη ρουτίνα σαν μία απλή local μεταβλητή.**

Π.χ.

```
procedure OneSingleParam ( a : integer );  
procedure TwoSameParams ( a,b : integer );  
function TwoDifParams ( a : byte; b : char ) : boolean;
```

Όπως φαίνεται από τα παραπάνω παραδείγματα, μπορούμε να δηλώσουμε δύο ή και περισσότερες παραμέτρους μαζί στην περίπτωση που έχουν ίδιο τ.δ. βάζοντας απλά κόμμα (,) ανάμεσα τους. Όταν έχουμε όμως παραμέτρους διαφορετικών τύπων τότε, μόλις γράψουμε τον τ.δ. βάζουμε ένα ερωτηματικό (;) και μετά συνεχίζουμε με τις υπόλοιπες παραμέτρους. Σημειώστε ότι πριν το κλείσιμο της παρένθεσης δε βάζουμε ερωτηματικό, δηλαδή μετά τον τ.δ. της τελευταίας παραμέτρου που δηλώνουμε.

Όταν καλούμε μία ρουτίνα με value parameters στη θέση των παραμέτρων μπορούμε να γράψουμε μία έκφραση του τύπου της παραμέτρου. Δηλαδή μία σταθερά, μία μεταβλητή, το αποτέλεσμα μίας συνάρτησης, πράξεις με όλα τα παραπάνω κ.λ.π.

Π.χ.

```
procedure TwoSameParams ( 55,a*Abs(i-12) );
```

Constant Parameters (Παράμετροι Const)

Στις δηλώσεις αυτών των παραμέτρων πριν από το όνομα τους γράφουμε τη λέξη const. Αυτό σημαίνει ότι η ρουτίνα θα πάρει την τιμή της παραμέτρου και θα τη χρησιμοποιήσει, αλλά δεν πρόκειται να την αλλάξει. **Η παράμετρος ενεργεί μέσα στη ρουτίνα σαν μία απλή local constant.**

Π.χ.

```
procedure PrintStars ( const n : byte );
```

Πρέπει να σημειωθεί ότι η παρουσίαση των constant parameters έγινε στην έκδοση 7.0 της Pascal, ενώ υπήρχαν εδώ και χρόνια στη γλώσσα C. Το ότι μέσα στη ρουτίνα η τιμή της παραμέτρου δε θα αλλάξει, δίνει τη δυνατότητα στον compiler να δημιουργήσει ισχυρότερο και γρηγορότερο κώδικα, ειδικά στις περιπτώσεις όπου η παράμετρος είναι τύπου string ή

άλλου σύνθετου τύπου (π.χ. array, record). Κατά τα άλλα θα λέγαμε ότι η λειτουργία const και απλών παραμέτρων είναι η ίδια.

Ενα άλλο σημείο που πρέπει να προσέξουμε είναι ότι αν έχουμε σκοπό μία παράμετρο να μην της αλλάξουμε τιμή μέσα στην procedure καλό είναι να την κάνουμε const. Αυτό θα μας προστατέψει από τυχόν λάθη (π.χ. κατά λάθος πάμε να την αλλάξουμε τιμή), αλλά θα μας βοηθήσει να καταλάβουμε μετά από καιρό βλέποντας μόνο την επικεφαλίδα της ρουτίνας, αν αλλάζουμε κάποια παράμετρο ή όχι.

Όταν καλούμε μία ρουτίνα με const parameter, ισχύουν όσα είπαμε και για τις value parameters. Δηλαδή, στη θέση της παραμέτρου μπορούμε να γράψουμε μία έκφραση του τύπου της παραμέτρου.

Variable Parameters (Παράμετροι Var)

Οι var παράμετροι, σε αντίθεση με τις δύο κατηγορίες που είδαμε προηγούμενα, έχουν σημαντική και ουσιαστική διαφορά. Στην περίπτωση αυτή στη ρουτίνα δεν έρχεται μία τιμή, αλλά το όνομα μίας μεταβλητής. Ότι αλλαγές γίνουν από τη ρουτίνα στην παράμετρο αυτή, θα έχουν αντίκτυπο στην πραγματική μεταβλητή που στείλαμε όταν καλέσαμε τη ρουτίνα.

Ουσιαστικά η παράμετρος αντιπροσωπεύει τη μεταβλητή που έχουμε στείλει.

Π.χ.

```
procedure Increase ( var a : integer );
```

Όταν καλούμε μία ρουτίνα που έχει var parameters πρέπει στη θέση της παραμέτρου αυτής να δώσουμε οπωσδήποτε το όνομα κάποιας μεταβλητής. Ο τύπος δεδομένων της μεταβλητής που στέλνουμε, πρέπει να είναι ακριβώς ίδιος με τον τύπο δεδομένων της παραμέτρου της procedure ή function.

Παράδειγμα

```
var
  a : integer;

procedure Double ( var n : integer );
begin
  n := n*2;
end;
```



```
begin
  a := 50;
  writeln ('Before procedure call : ',a);
  Double (a);
  writeln ('After procedure call   : ',a);
end.
```

Στο παραπάνω πρόγραμμα η procedure Double διπλασιάζει την τιμή της μεταβλητής που της στέλνουμε. Ετσι το πρόγραμμα θα τυπώσει:

```
Before procedure call : 50
After procedure call   : 100
```

γιατί ενώ κάναμε τη μεταβλητή μας (a) ίση με 50, αφού τη στείλαμε στην procedure Double, αυτή διπλασιάστηκε. Ετσι μετά την κλήση της procedure η μεταβλητή μας έχει γίνει 100.

ΚΕΦΑΛΑΙΟ 7

Strings

Ο τύπος δεδομένων **string** είναι πάρα πολύ συνηθισμένος στην Pascal και γενικά σε όλες τις γλώσσες προγραμματισμού. Για το λόγο αυτό συναντούμε διάφορες υπορουτίνες της γλώσσας που είναι γραμμένες για χειρισμό strings.

Θα ήταν καλό να υπενθυμίσουμε μερικά στοιχεία για τα strings. Για να ορίσουμε μια μεταβλητή τύπου string γράφουμε στο variable declaration :

```
var
  OneString      : string;
  AnotherString  : string[40];
```

Η μεταβλητή OneString θα έχει μέγιστο μήκος 255 χαρακτήρες ενώ η AnotherString 40 χαρακτήρες. Λέγοντας μέγιστο μήκος εννοούμε ότι δεν μπορούμε να καταχωρήσουμε στη μεταβλητή μας ένα string (αλφαριθμητικό) με περισσότερους χαρακτήρες από αυτόν τον αριθμό.

Η αλφαριθμητική σταθερά είναι και αυτή ένα string που περικλείεται όμως ανάμεσα σε δύο εισαγωγικά ('). Σε περίπτωση που θέλουμε να καταχωρήσουμε και τον χαρακτήρα ' μέσα στα εισαγωγικά τότε γράφουμε δύο τέτοιους χαρακτήρες τον ένα δίπλα στον άλλο. Για παράδειγμα δείτε τι τυπώνουν οι παρακάτω δύο προτάσεις writeln :

```
writeln ('This is a string'); -> This is a string.
writeln ('It's a string');   -> It's a string
```

Μερικά σημεία ακόμα πρέπει να προσεχθούν ιδιαίτερα όταν έχουμε να χειριστούμε strings και αυτά είναι τα παρακάτω :

- OneString := AnotherString;

Αν το μέγιστο μήκος του OneString είναι μικρότερο από το αληθινό μήκος του AnotherString, τότε το OneString κρατάει μόνο τους χαρακτήρες που μπορεί να κρατήσει. Αν δηλαδή είχαμε δηλώσει τη μεταβλητή OneString σαν string[4] και το AnotherString είναι ίσο με 'computer' τότε το OneString θα γίνει ίσο με 'comp'.

- `OneString := OneChar;`

Είναι δυνατόν να εξισώσουμε μια μεταβλητή `string` με μια μεταβλητή `char` ή ακόμα να προσθέσουμε `chars` με `strings` και να πάρουμε αποτελέσματα `strings`.

- `OneChar := OneString[5];`

Σε περίπτωση που χρειαζόμαστε για οποιονδήποτε λόγο έναν μόνο χαρακτήρα ενός `string`, τότε γράφουμε τον αριθμό του χαρακτήρα μέσα σε αγκύλες `[]` δίπλα ακριβώς από το όνομα της μεταβλητής. Στο παραπάνω παράδειγμα δίνουμε τον πέμπτο χαρακτήρα του `OneString` στο `OneChar`.

Procedures & Functions για Χειρισμό Strings

- **function Length (s : string) : byte;**

Επιστρέφει το μήκος του `s` σε χαρακτήρες.

```
var
    OneString : string;

begin
    write ('Δώσε ένα string : ');
    readln (OneString);
    writeln ('Έχει μήκος ',Length(s),' χαρακτήρες. ');
    readln;
end.
```

- **function Pos (substr,s : string) : byte;**

Η συνάρτηση αυτή ψάχνει για το `substr` μέσα στο `s`, και αν υπάρχει επιστρέφει τη θέση που βρίσκεται (από ποιόν χαρακτήρα του `s` ξεκινάει). Αν δεν υπάρχει τότε επιστρέφει τον αριθμό 0.

```
begin
    writeln (Pos('SERVE','OBSERVE'));
end.
```

Το παραπάνω παράδειγμα τυπώνει τον αριθμό 3.

- **function Copy (s : string; index,count : integer) : string;**

Επιστρέφει το κομμάτι του `s` που αρχίζει από τη θέση `index` και έχει μήκος `count` χαρακτήρες.

```
begin
    writeln (Copy('COMPUTER',4,3));
end.
```

Το παραπάνω παράδειγμα τυπώνει τη λέξη 'PUT', από τον 4ο χαρακτήρα, τρεις χαρακτήρες (4ο, 5ο και 6ο).

- **function Concat (s1 [, s2, ..., sn] : string) : string;**

Κάθε παράμετρος της Concat είναι string. Όπως φαίνεται στη σύνταξη μπορούμε να βάλουμε από μία μέχρι όλες παραμέτρους θέλουμε (οι αγκύλες [] δηλώνουν ότι είναι προαιρετικές οι μεταβλητές που περικλείουν). Το αποτέλεσμα της συνάρτησης είναι ένα string που έχει συνενώσει όλες τις παραμέτρους. Ακόμα μπορούμε να χρησιμοποιήσουμε και το (+), για να έχουμε το ίδιο αποτέλεσμα με την Concat.

```
var
  S1,S2 : string;

begin
  S1 := Concat ('ABC','DEF');
  S2 := 'ABC'+ 'DEF';
end.
```

Στο παραπάνω παράδειγμα οι μεταβλητές S1,S2 είναι ίσες μεταξύ τους και μάλιστα είναι και ίσες με 'ABCDEF'.

- **procedure Insert (s1 : string; var s : string; index : integer);**

Το s1 είναι μια αλφαριθμητική έκφραση (μεταβλητή ή σταθερά), ενώ το s είναι μια αλφαριθμητική μεταβλητή. Το index είναι μια ακέραια έκφραση (μεταβλητή ή σταθερά). Η procedure Insert εισάγει το s1 μέσα στο s, στη θέση index.

```
var
  s : string;

begin
  s := 'tm';
  Insert ('ea',s,2);
  writeln (s);
end.
```

Το παραπάνω πρόγραμμα τυπώνει τη λέξη 'team'.

- **procedure Delete (var s : string; index,count : integer);**

Το s είναι μια μεταβλητή τύπου string και τα index,count είναι δύο ακέραιες μεταβλητές ή σταθερές. Η procedure Delete σβήνει το κομμάτι του s που αρχίζει από τη θέση index και έχει count χαρακτήρες.

```
var
  s : string;

begin
  s := 'Turbo Pascal';
  Delete (s,2,5);
  writeln (s);
end.
```

Το παραπάνω πρόγραμμα τυπώνει 'TPascal'.

- **ΠΑΡΑΤΗΡΗΣΕΙΣ**

Αν το index είναι μεγαλύτερο από το μέγεθος του s δεν σβήνονται χαρακτήρες. Αν το count είναι μεγαλύτερο από το υπόλοιπο του s (που αρχίζει από τη θέση index, δηλαδή $\text{count} > \text{Length}(s) - \text{index}$) τότε σβήνεται όλο το υπόλοιπο του s.

- **procedure Str (x [:width [:decimals]]; var s :string);**

Το x είναι μία ακέραια ή πραγματική έκφραση, τα width και decimals είναι ακέραιες εκφράσεις, και το s είναι μια μεταβλητή τύπου string. Η Str μετατρέπει το x σε αλφαριθμητικό s, χρησιμοποιώντας width χαρακτήρες και decimals δεκαδικά ψηφία. Όπως στην writeln γράφουμε έναν αριθμό χρησιμοποιώντας width και decimals, π.χ.

```
writeln (AReal:5:2);
```

έτσι και στην Str μπορούμε να πούμε πόσους συνολικά χαρακτήρες θα καταλάβει το x και πόσα δεκαδικά ψηφία θα έχει.

```
var
  R      : real;
  I      : integer;
  s1,s2  : string;
begin
  R := 4.5;
  I := 78;
  Str (I:5,S1);
  Str (R:5:2,S2);
  writeln ('*',S1,'*');
  writeln ('*',S2,'*');
end.
```

Το παραπάνω παράδειγμα θα τυπώσει :

```
*   78*
* 4.50*
```

(Τα αστεράκια τα βάζουμε για να δούμε και πρακτικά τα κενά που τυπώνονται).

ΠΑΡΑΤΗΡΗΣΗ. Στο σημείο αυτό πρέπει να παρατηρήσουμε ότι η procedure Str λόγω του ότι είναι procedure είναι ίσως πολλές φορές άβολη. Η γλώσσα BASIC π.χ. είχε τη συνάρτηση STR\$ η οποία έκανε τη μετατροπή. Στην pascal είναι πολύ εύκολο όμως να κάνουμε και εμείς μία τέτοια συνάρτηση, έτσι ώστε να κάνουμε την μετατροπή από αριθμό σε string πιο άμεσα (φτιάξτε την μόνοι σας για εξάσκηση). Μάλιστα, είναι γεγονός ότι στο νέο προϊόν της Borland (Delphi) υπάρχει μία τέτοια συνάρτηση με το όνομα IntToStr.

- **procedure Val (s : string; var v; var code : integer);**

Το s είναι string, το v είναι μια ακέραια ή πραγματική μεταβλητή, και το code είναι μια ακέραια μεταβλητή. Η Val μετατρέπει το s στον αριθμό v. Αν το string s, είναι για κάποιο τρόπο αδύνατο να μετατραπεί σε αριθμό τότε η μεταβλητή code γίνεται διαφορετική από 0, όταν όμως η μετατροπή γίνεται χωρίς πρόβλημα τότε το code είναι 0.

ΠΡΟΣΟΧΗ : Πρέπει το s να μην έχει κενά στο τέλος!

```
var
  i,code   : integer;
  s        : string;

begin
  s := '3223';
  Val (s,i,code);
  writeln ('Αριθμός           : ',i);
  writeln ('Αποτέλεσμα μετατροπής : ',code);
  readln;
end.
```

Στο παραπάνω παράδειγμα το i είναι ίσο με 3223, και το code είναι ίσο με 0. Αλλάξτε το s, και δείτε τα αποτελέσματα!

Περίληψη String Procedures & Functions

- **Functions**

- Concat Προσθέτει (ενώνει) μια σειρά από strings.
- Copy Επιστρέφει ένα μέρος (κομμάτι) από ένα string.
- Length Επιστρέφει το μήκος του string σε χαρακτήρες.
- Pos Ψάχνει για ένα string μέσα σε ένα άλλο string.

- **Procedures**

- Delete Σβήνει ένα κομμάτι από ένα string.
- Insert Εισάγει ένα string μέσα σε ένα άλλο string.
- Str Μετατρέπει έναν αριθμό σε string.
- Val Μετατρέπει ένα string σε αριθμό.

Γενικά για τα Strings

Ολες οι παραπάνω procedures & functions για string handling ανήκουν στη Standard Pascal και είναι οι άκρως απαραίτητες. Αυτό σημαίνει ότι ίσως μας φανούν λίγες και ανεπαρκείς. Η Pascal όμως μας δίνει τη δυνατότητα να φτιάξουμε δικές μας procedures & functions, με αποτέλεσμα οι δυνατότητες χειρισμού strings να είναι απεριόριστες ...

Καλά παραδείγματα για εξάσκηση, είναι συναρτήσεις όπως οι Left, Right, IntegerToString, RealToString με προφανή αποτελέσματα.

Στους δομημένους τύπους δεδομένων της Pascal ανήκουν και οι **πίνακες (arrays)**. Χαρακτηριστικά ενός πίνακα, είναι το γεγονός ότι μπορεί να θεωρηθεί σαν μία μεταβλητή, ενώ στην πραγματικότητα είναι ένα σύνολο από πολλές μεταβλητές του ίδιου τύπου δεδομένων. Ακόμα για να διαχειριστούμε ολόκληρο τον πίνακα, χρειαζόμαστε μόνο το όνομά του, ενώ αν θέλουμε να κάνουμε διαχείριση των στοιχείων του πίνακα τότε χρειαζόμαστε και έναν δείκτη που είναι ουσιαστικά ο "αριθμός σειράς" του κάθε στοιχείου.

Η έννοια ενός πίνακα (array) δίνεται καλύτερα με ένα παράδειγμα. Ας δούμε το πρόβλημα της ανάγνωσης πέντε χαρακτήρων και της παρουσίασής τους με αντίστροφη σειρά. Εύκολα μπορεί να πεί κανείς ότι εφόσον θα διαβάσουμε πέντε χαρακτήρες, χρειαζόμαστε και πέντε μεταβλητές τύπου char. Διαβάζουμε με τη σειρά τους χαρακτήρες τους βάζουμε με τη σειρά στις μεταβλητές μας και ύστερα τους τυπώνουμε με την αντίθετη σειρά από αυτή που τους καταχωρήσαμε :

```
var
  c1,c2,c3,c4,c5 : char;
begin
  readln (c1,c2,c3,c4,c5);
  writeln (c5,c4,c3,c2,c1);
end.
```

Αν όμως το πρόγραμμα αυτό έπρεπε να αντιστρέφει είκοσι χαρακτήρες, θα αρχίζαμε να έχουμε προβλήματα! Εξετάζοντας το πρόγραμμα, όμως, παίρνουμε μια ιδέα για το τι χρειάζεται για να λυθεί το πρόβλημα : Αν μπορούσαμε να γράψουμε ci όπου i είναι μια ακέραια μεταβλητή, ώστε να επιλέγουμε μια από τις 5 μεταβλητές c1 έως c5, θα μπορούσαμε να διαβάσουμε τους χαρακτήρες με for και να τους τυπώσουμε με μια ανάποδη for. Με τους πίνακες λοιπόν μπορούμε να θεωρήσουμε την ci σαν μια σειρά πέντε μεταβλητών. Το όνομα της σειράς είναι c και η κάθε μεταβλητή αναγνωρίζεται από τον αριθμό σειράς (i).

Στην Pascal η ομάδα αυτή των μεταβλητών ονομάζεται πίνακας (array). Η μεταβλητή ή σταθερά που χρειάζεται για την επιλογή μιας συγκεκριμένης μεταβλητής από τον πίνακα ονομάζεται δείκτης. Για να ορίσουμε ένα πίνακα γράφουμε στις δηλώσεις var :

όνομα πίνακα : **array**[περιοχή ordinal] **of** τύπος δεδομένων ;

δηλαδή για να ορίσουμε έναν πίνακα με το όνομα MyArray που να έχει συνολικά 20 θέσεις με ακέραιους αριθμούς γράφουμε :

```
var
  MyArray : array[1..20] of integer;
```

Για να δηλώσουμε ότι χρησιμοποιούμε μια μεταβλητή (έναν integer δηλαδή) από αυτόν τον πίνακα χρειάζεται να γράψουμε και τον δείκτη της θέσης αυτής της μεταβλητής. Π.χ. με την πρόταση :

```
writeln (MyArray[5]);
```

λέμε στην Pascal να τυπώσει τον πέμπτο ακέραιο (την πέμπτη στη σειρά μεταβλητή του πίνακά μας!

Για τη λύση τώρα του προηγούμενου προβλήματός μας έχουμε :

```
var
  c : array[1..5] of char;
  i : byte;
begin
  for i := 1 to 5 do
    readln (c[i]);
  for i := 5 downto 1 do
    write (c[i]);
end.
```

Διαχείριση Πινάκων

- Πρόβλημα

Να ορίσετε ένα πίνακα με 20 στοιχεία, που κάθε ένα από αυτά να είναι ένας αριθμός τύπου byte. Να διαβάσετε όλα τα στοιχεία του πίνακα από το πληκτρολόγιο αφού πρώτα τυπώνετε τον αριθμό σειράς του στοιχείου που περιμένετε να γράψει ο χρήστης. Κατόπιν να καθαρίσετε την οθόνη και να τυπώσετε τα στοιχεία του πίνακα κατακόρυφα.

Αφού περιμένετε να πατηθεί το πλήκτρο Enter να γεμίσετε τον πίνακα με τυχαίους αριθμούς από το 1 έως το 200 και να τυπώσετε τον πίνακα σε μία γραμμή.

```
uses
  Crt;
var
  Pin : array[1..20] of byte;      ορισμός πίνακα
  i : byte;
begin
  ClrScr;
  for i := 1 to 20 do
    begin
      write (i:4, '>');
      readln (Pin[i]);
    end;
  ClrScr;
```

Διάβασμα στοιχείων του πίνακα. Σε κάθε γραμμή τυπώνουμε τον αριθμό του στοιχείου, ένα > και περιμένουμε με readln να "πάρουμε" το νούμερο που θα δώσει ο χρήστης.

<pre>for i := 1 to 20 do writeln (Pin[i]:4);</pre>	Τυπώνουμε τον πίνακα κάθετα. Χρησιμοποιούμε τη writeln γι' αυτό. Το :4 το βάζουμε για "ομορφιά", για να είναι τα νούμερα στοιχισμένα δεξιά δηλαδή. Εφόσον τα νούμερα είναι bytes, δηλαδή το πολύ 3 χαρακτήρες το :4 είναι σίγουρο ότι βολεύει.
<pre>readln;</pre>	Το readln μπαίνει για να καθυστερήσουμε την εκτέλεση του προγράμματος. Θα περιμένει να πατηθεί Enter.
<pre>for i := 1 to 20 do Pin[i] := Random(200)+1; ClrScr; for i := 1 to 20 do write (Pin[i]:4);</pre>	Σε κάθε στοιχείο βάζουμε δικές μας τιμές, τυχαίες από 1 έως 200.
<pre>readln; end.</pre>	Τυπώνω τον πίνακα οριζόντια. Χρησιμοποιούμε τη write για να τυπώνει τα νούμερα το ένα δίπλα στο άλλο. Το :4 εκτός από "ομορφιά" είναι και λειτουργικό για να μην "κολλήσουν" τα νούμερα.

Πίνακες Δύο Διαστάσεων (Διδιάστατοι)

Οι πίνακες που μέχρι τώρα συζητήθηκαν ονομάζονται μονοδιάστατοι γιατί χρειαζόμαστε ένα μόνο δείκτη για να αντιπροσωπεύσουμε ένα στοιχείο τους. Σκεφτείτε τώρα το παρακάτω πρόβλημα :

Χρειαζόμαστε κάποιες μεταβλητές για να καταχωρήσουμε τους βαθμούς 20 μαθητών (τους έχουμε αριθμήσει από 1 έως 20) σε τέσσερα tests (και αυτά αριθμημένα από 1 έως 4). Με βάση αυτά που ξέρουμε μπορούμε να πούμε ότι για έναν μόνο μαθητή θα μπορούσαμε να έχουμε 1 πίνακα με 4 στοιχεία. Σωστά! Τώρα που έχουμε 20 μαθητές;

Η λύση είναι πολύ εύκολη : Χρειαζόμαστε έναν "πίνακα από πίνακες" των 4 στοιχείων. Για να το πούμε πιο αναλυτικά : *Εναν πίνακα 20 θέσεων που κάθε μία αντιπροσωπεύει ένα μαθητή, και σε κάθε θέση του πίνακα αυτού θα έχουμε ένα πίνακα 4 θέσεων που είναι ουσιαστικά οι βαθμοί του μαθητή στα τέσσερα tests.*

Για να το πούμε αυτό τώρα σε Pascal γράφουμε :

```
var
  StudentTest : array[1..20] of array[1..4] of byte;
```

Ερώτηση : πόσους δείκτες χρειαζόμαστε για να μάθουμε το βαθμό ενός μαθητή σε κάποιο test;

Απάντηση : μα φυσικά δύο! Εναν για να ξέρουμε πιο μαθητή θέλουμε και έναν για να ξέρουμε το test!

Με λίγα λόγια στον παραπάνω ορισμό της Pascal, η πρώτη περιοχή 1..20 είναι η περιοχή των μαθητών και η δεύτερη 1..4 η περιοχή των tests. Με την μεταβλητή **StudentTest[5][2]** εννοούμε το βαθμό του πέμπτου μαθητή στο δεύτερο test.

Για να διαβάσουμε τώρα από το πληκτρολόγιο στοιχεία για τον παραπάνω πίνακα χρησιμοποιούμε την παρακάτω ρουτίνα :

```
var
  StudentTest : array[1..20] of array[1..4] of byte;
  i,j          : byte;
begin
  for i := 1 to 20 do
    begin
      writeln ('Μαθητής ',i,' : '); {π.χ. Μαθητής 5 : }
      for j := 1 to 4 do
        begin
          write ('Test ',j,' : ');    {π.χ. Test 3 : }
          readln (StudentTest[i][j]);
        end;
      end;
    end.
end.
```

Με το παραπάνω πρόγραμμα θα έχουμε το εξής αποτέλεσμα :

```
Μαθητής 1 :
Test 1 : 19
Test 2 : 17
Test 3 : 15
Test 4 : 10
Μαθητής 2 :
Test 1 : 13
Test 2 : 14
```

Κ.Ο.Κ.

Όπως βλέπετε για να χειριστούμε έναν δισδιάστατο πίνακα χρειαζόμαστε δύο for το ένα μέσα στο άλλο. Το ένα for μετράει τις τιμές του πρώτου δείκτη (1..20) και το δεύτερο τις τιμές του δεύτερου δείκτη (1..4). Έτσι το στοιχείο StudentTest[i][j] αντιπροσωπεύει τον i μαθητή και το j test.

Την δήλωση του παραπάνω πίνακα δύο διαστάσεων μπορούμε να την κάνουμε και ως εξής :

```
var
  StudentTest : array[1..20,1..4] of byte;
```

και την επεξεργασία των στοιχείων του σαν StudentTest[i,j]. Δεν υπάρχει καμία διαφορά στους δύο τρόπους ορισμού του πίνακα, όπως και στους δύο τρόπους διαχείρισης των στοιχείων του. Για την ακρίβεια μπορούμε να δηλώσουμε τον πίνακα με τον πρώτο

τρόπο και να τον χρησιμοποιούμε με τον δεύτερο! Η Pascal δέχεται και τα δύο σαν το ίδιο οπότε γενικά αυτό που θυμόμαστε είναι ότι ΔΕΝ υπάρχει περιορισμός!

- **Πρόβλημα**

Θέλουμε να καταχωρήσουμε στον υπολογιστή τις νικήτριες στήλες του δελτίου ΠΡΟ-ΠΟ των τελευταίων πέντε δελτίων. Επειδή το ΠΡΟ-ΠΟ χρησιμοποιεί τα στοιχεία 1-X-2, κάθε σημείο θα πρέπει να είναι του τύπου char. Να γίνει πρόγραμμα που θα ζητάει από το πληκτρολόγιο τα δεδομένα αυτά και θα τα καταχωρεί στη μνήμη του υπολογιστή. Υστερα αφού καθαρίσει την οθόνη θα τα τυπώνει.

Θα χρησιμοποιήσουμε ένα πίνακα 5 θέσεων (για τις πέντε αγωνιστικές) όπου σε κάθε θέση θα υπάρχει ένας πίνακας 13 θέσεων με chars.

```
var
  ProPo : array[1..5,1..13] of char;
  i,j   : byte;
begin
  for i := 1 to 5 do
  begin
    writeln ('Δελτίο ',i,' : ');
    for j := 1 to 13 do
    begin
      write ('Σημείο ',j,' : ');
      readln (ProPo[i][j]);
    end;
  end;
  ClrScr;
  for i := 1 to 5 do
  begin
    for j := 1 to 13 do
      write (ProPo[i][j]);
    writeln;
  end;
end.
```

Δήλωση του πίνακα και των βοηθητικών μεταβλητών του προγράμματος. Προσέξτε ότι θα μπορούσαμε να πούμε και array[1..5] of array[1..13] of char; Το i μετράει τα δελτία και το j τα σημεία. Διαβάζω λοιπόν από το πληκτρολόγιο το j σημείο του δελτίου i.

Τυπώνω τον πίνακα. Στο loop των σημείων έχω write για να τυπώνονται το ένα δίπλα στο άλλο, αμέσως μετά όμως (αφού τυπωθούν και τα 13) βάζουμε ένα writeln για να "πάμε" στην επόμενη γραμμή.

Το παραπάνω πρόγραμμα θα εκτυπώσει τις νικήτριες στήλες ως εξής :

```
1111X1XX12112  Δηλαδή κάθε στήλη σε μια γραμμή της οθόνης.
1111111111XXX
X1X11X11222X2
X11211121X12X
1X1112X21X221
```

Αν θέλαμε να τα τυπώσουμε όπως είναι στην πραγματικότητα τότε θα έπρεπε να αντιστρέψουμε τα δύο for. Δηλαδή εξωτερικά θα είχαμε το

for j := 1 to 13 do και μέσα σ'αυτό το **for i := 1 to 5 do**. Δοκιμάστε το.

Γενικά για τους Πίνακες

Οι πίνακες είναι ο πιο συνηθισμένος δομημένος τύπος δεδομένων που βρίσκουμε σε όλα τα προγράμματα. Είναι καλό να ξέρει κανείς να χρησιμοποιεί ειδικά πίνακες, μια και πάρα πολλά προβλήματα λύνονται με τη βοήθειά τους. Εκτός από τους δισδιάστατους πίνακες έχουμε και τρισδιάστατους και γενικά πολυδιάστατους αλλά εκτός του ότι αυτές οι περιπτώσεις είναι σπάνιες είναι και δύσκολο να φανταστούμε πως είναι ένας τετραδιάστατος πίνακας π.χ. Το μόνο που μπορούμε να πούμε είναι ότι ένας τρισδιάστατος πίνακας είναι ουσιαστικά πολλοί δισδιάστατοι, και ένας τετραδιάστατος πίνακας είναι πολλοί τρισδιάστατοι κ.ο.κ.

Ενα πολύ σημαντικό πράγμα που δεν πρέπει να ξεχνάμε ποτέ, είναι η σειρά των δεικτών indexes. Όπως έχουμε ορίσει τον πίνακα, έτσι θα πρέπει να τον χρησιμοποιούμε. Επειδή πολλές φορές στους διδιάστατους μπερδευόμαστε, ειδικά όταν έχουμε ίσο αριθμό στηλών και γραμμών, έχει επικρατήσει ο πρώτος δείκτης να δείχνει τις γραμμές και ο δεύτερος τις στήλες. Γραμμή - στήλη, λοιπόν

Sets (Σύνολα)

Τα sets (σύνολα) είναι ένας πάρα πολύ σημαντικός τύπος δεδομένων που υπάρχει μόνο στην Turbo Pascal. Ένα set ορίζεται σαν ένα σύνολο από στοιχεία ενός τύπου δεδομένων.

set type ----> set of ordinal type

Ο τύπος δεδομένων είναι πάντα **ordinal** και πρέπει οπωσδήποτε οι τιμές του να είναι μέχρι 256. Αυτό σημαίνει ότι στα sets μπορούμε να χρησιμοποιήσουμε μόνο τους τύπους byte, char και τα enumerated που φτιάχνουμε μόνοι μας.

Παραδείγματα Δηλώσεων και Χρήσης Sets

Παρακάτω φαίνονται δηλώσεις συνόλων, είτε σαν typed constants είτε σαν μεταβλητές.

```
const
  X : set of char = ['A'..'F'];
var
  B : set of byte;
```

Μέσα στον κώδικα του προγράμματος οι παρακάτω προτάσεις δίνουν τιμές σε sets.

```
X := [];           { κενό σύνολο }
B := [34,39];      { περιέχει τους αριθμούς 34 και 39}
X := ['A'..'Z'];    { περιέχει τους χαρακτήρες από 'A' μέχρι και 'Z' }
```

Είναι φανερό ότι στα sets χρησιμοποιούμε τους χαρακτήρες '[' και ']', για να περικλείσουμε τα στοιχεία τους. Τα στοιχεία δε, μπορεί να γράφονται μεμονωμένα χωρισμένα με κόμμα ή σαν περιοχές τιμών χωρισμένα με δύο τελείες (..) ή και ανακατωμένα π.χ. ['0'..'9','A'..'Z','a'..'z','\$','%']

Πράξεις με Sets

Στο Κεφάλαιο των operators (τελεστών) είδαμε κάποιες πράξεις που μπορούν να γίνουν με τα sets. Παρακάτω είναι μια λίγο πιο λεπτομερής αναφορά. Να σημειωθεί ότι τα sets είναι συγκρίσιμοι τύποι στην Pascal και μπορούμε να δούμε αν δύο sets είναι ίσα, αν το ένα είναι υποσύνολο του άλλου κ.λ.π.

Ένωση	$A + B$	Περιέχει τα στοιχεία και των δύο συνόλων. (σημ. τα κοινά στοιχεία των δύο συνόλων βρίσκονται μία φορά στο $A+B$)
Τομή	$A * B$	Περιέχει τα κοινά τους στοιχεία.
Διαφορά	$A - B$	Περιέχει τα στοιχεία του A εκτός από αυτά που περιέχονται και στο B.
Ανήκει	$c \text{ in } A$	Είναι true όταν το στοιχείο c ανήκει στο σύνολο A.
Ισότητα	$A = B$	Είναι true όταν τα A και B περιέχουν ακριβώς τα ίδια στοιχεία. Σε κάθε άλλη περίπτωση $A \neq B$.
Υποσύνολο	$A \leq B$	Είναι true όταν όλα τα στοιχεία του A περιέχονται και στο B.
Υπερσύνολο	$A \geq B$	Είναι true όταν όλα τα στοιχεία του B περιέχονται και στο A.

Σημείωση. Από τα μαθηματικά ξέρουμε επίσης και τις έννοιες **γνήσιο υποσύνολο** και **γνήσιο υπερσύνολο**. Αν το A είναι γνήσιο υποσύνολο του B, συνεπάγεται ότι όλα τα στοιχεία του A περιέχονται στο B, αλλά το B έχει και κάποια παραπάνω. Δεν είναι δηλαδή ίσα. Στην Turbo Pascal όπως φαίνεται παραπάνω δεν υποστηρίζεται ο έλεγχος της γνησιότητας ενός υποσυνόλου. Αμα θέλουμε όμως μπορούμε πολύ απλά και κατανοητά να πούμε :

```
if (A <= B) and (A <> B) then
  writeln ('Το A είναι γνήσιο υποσύνολο του B.');
```

Παραδείγματα

Εστω οι παρακάτω δηλώσεις :

```
const
  A : set of char = ['A'..'Z'];
  B : set of char = ['B','O','R','L','A','N','D'];
  C : set of char = ['T','U','R','B','O'];
```

τότε :

Η πράξη...

...είναι ίση με :

$B+C$	<code>['B','O','R','L','A','N','D','T','U']</code>
$A+B$	<code>['A'..'Z']</code>
$B-C$	<code>['L','A','N','D']</code>
$B*C$	<code>['B','O','R']</code>
$B \leq A$	true
$B \geq C$	false
<code>'K' in B</code>	false
$(A+B) = A$	true

Χρήση Συνόλων (Sets)

Για να προσθέσουμε την μεταβλητή *c* (char) σε ένα σύνολο *X* (set of char) χρησιμοποιούμε την ένωση συνόλων : $X := X + [c]$; Βάζουμε δηλαδή το *c* μέσα σε αγκύλες για να δηλώσουμε ότι έχουμε ένα set στο οποίο ανήκει μόνο το *c*.

Τα sets χρησιμεύουν ιδιαίτερα σε περιπτώσεις που θέλουμε κάποια μεταβλητή να πάρει κάποια συγκεκριμένη τιμή. Η χρήση τους βοηθά πολύ στην αναγνωσιμότητα των προγραμμάτων μας ενώ παράλληλα γλιτώνουμε και γράψιμο. Στο παρακάτω παράδειγμα, μετράμε τους χαρακτήρες ενός string που είναι γράμματα του λατινικού αλφαβήτου με τον κλασσικό τρόπο:

```
n := 0;
for i := 1 to Length(s) do
  if ((s[i] >= 'A') and (s[i] <= 'Z')) or
    ((s[i] >= 'a') and (s[i] <= 'z')) then
    Inc (n);
```

Ο τρόπος με τα sets είναι πολύ πιο ευανάγνωστος και μικρός:

```
n := 0;
for i := 1 to Length(s) do
  if s[i] in ['A'..'Z', 'a'..'z'] then
    Inc (n);
```

Για να εισάγουμε ένα στοιχείο *x* σε ένα set *S* μπορούμε να γράψουμε $S := S + [x]$; και για να το βγάλουμε από το σύνολο $S := S - [x]$; Ουσιαστικά περικλείουμε το *x* μέσα σε αγκύλες έτσι ώστε να θεωρηθεί μονοσύνολο που περιέχει μόνο το *x*, και ύστερα κάνουμε ένωση ή διαφορά του μονοσυνόλου με το σύνολο *S*. Η Turbo Pascal 7.0 όμως, μας δίνει δύο procedures για να κάνουμε αυτή τη δουλειά καλύτερα. Και το κυριότερο, πιο γρήγορα :

- **procedure Exclude (var *S* : set of type T; *i* : type T);**
- **procedure Include (var *S* : set of type T; *i* : type T);**

Το *S* είναι μία μεταβλητή set κάποιου τύπου δεδομένων και το *i* είναι μία έκφραση αυτού του τύπου. Η Exclude βγάζει από το *S*, το *i* ενώ η Include συμπεριλαμβάνει στο *S*, το *i*.

Ενα record στην Pascal είναι ένας δομημένος τύπος δεδομένων, που αποτελείται από δεδομένα διαφορετικών τύπων. Σε αντίθεση με τους πίνακες που είναι ομάδες από στοιχεία του ίδιου τύπου (π.χ. array[...] of byte), ένα record είναι ομάδα στοιχεία διαφορετικού τύπου (π.χ. ένα byte, ένα string και ένα boolean).

Κάθε στοιχείο ενός record ονομάζεται πεδίο (field) και μπορεί να είναι οποιουδήποτε τύπου δεδομένων.

Δηλώσεις Records

Για να ορίσουμε μία μεταβλητή τύπου record γράφουμε :

```
record
    όνομα πεδίου : τύπος δεδομένων ;
    .
    .
end;
```

Το παρακάτω είναι ένα παράδειγμα μεταβλητής τύπου record με τρία πεδία, Έτος, Μήνας και Ημέρα.

```
var
    MyRec : record
        Year   : integer;
        Month  : byte;
        Day    : byte;
    end;
```

Στο προηγούμενο παράδειγμα η δήλωση του record έγινε στον τομέα των δηλώσεων μεταβλητών. Συνήθως όμως τα records τα δηλώνουμε στον τομέα των δηλώσεων τύπων δεδομένων (**type**). Ουσιαστικά δηλαδή δίνουμε ένα όνομα σε κάθε τύπο record που φτιάχνουμε και ακολούθως χρησιμοποιούμε το όνομα αυτό για τις δηλώσεις. Αυτό βολεύει πάρα πολύ (πρακτικά είναι και αναπόφευκτο) όταν χρησιμοποιούμε τα records στα αρχεία.

Ετσι η δήλωση του προηγούμενου record θα μπορούσε να γίνει ξανά κάπως έτσι:

```
type
```



```

RecType =   record
              Year   : integer;
              Month   : byte;
              Day     : byte;
            end;

var
  MyRec : RecType;

```

Για να δώσουμε τιμές στα πεδία του παραπάνω record θα πρέπει να γράψουμε το όνομα της μεταβλητής (MyRec), δίπλα μία τελεία (.) και δίπλα το όνομα κάθε πεδίου. Δηλαδή :

```

begin
  MyRec.Year := 1995;
  MyRec.Month := 4;
  MyRec.Day := 3;
end.

```

Πρέπει να προσέξουμε την περίπτωση που χρησιμοποιούμε το record μόνο του (π.χ. MyRec := AnotherRec;) και την περίπτωση που χρησιμοποιούμε το record μαζί με τα πεδία του. Ετσι λέγοντας MyRec εννοούμε ολόκληρο το record, ενώ λέγοντας MyRec.Year εννοούμε μόνο το πρώτο πεδίο του record το οποίο μάλιστα είναι του τύπου integer άρα μπορούμε χωρίς κανένα πρόβλημα να το χρησιμοποιήσουμε σαν να ήταν μια κανονική μεταβλητή τύπου integer.

Χειρισμός Records - Πρόταση with

Την πρόταση with τη γνωρίσαμε στο κεφάλαιο των προτάσεων (statements). Στο κεφάλαιο αυτό θα την περιγράψουμε με λίγο μεγαλύτερη λεπτομέρεια. Η πρόταση with είναι πρακτικά ένας άλλος τρόπος για να χειριστούμε τα records και τα πεδία τους. Η σύνταξη της είναι :

with μεταβλητή τύπου record do
 πρόταση ;

Με την πρόταση αυτή μπορούμε να χρησιμοποιήσουμε τα πεδία ενός record χωρίς να χρειάζεται να γράψουμε και το όνομα του record πριν από κάθε πεδίο. Ας δούμε καλύτερα ένα παράδειγμα με τον τρόπο που περιγράψαμε πιο πάνω και με την πρόταση with.

```

var
  aDate =   record
              Year   : integer;
              Month   : byte;
              Day     : byte;
            end;

```

```

begin
  write ('Δώσε ημερομηνία γράφοντας Ημέρα Μήνα Χρόνο : ');
  readln (aDate.Day,aDate.Month,aDate.Year);
  writeln;
  writeln (aDate.Day,'/',aDate.Month,'/',aDate.Year);
end.

```

Το προηγούμενο κυρίως πρόγραμμα μπορεί να γραφεί και ως εξής :

```

begin
  write ('Δώσε ημερομηνία γράφοντας Ημέρα Μήνα Χρόνο : ');
  with aDate do
    readln (Day,Month,Year);
    writeln;
  with aDate do
    writeln (Day,'/',Month,'/',Year);
  end.
end.

```

Ακόμα είναι προφανές ότι μπορούμε στην πρόταση with να γράψουμε και σύνθετη πρόταση (begin - end;). Δηλαδή το παρακάτω είναι σωστό :

```

with aDate do
begin
  Day := 15;
  Month := 5;
  Year := 1994;
end;

```

Πρέπει να σημειώσουμε ότι "μέσα" σε μια πρόταση with, προτεραιότητα έχουν τα πεδία και όχι οι υπόλοιπες μεταβλητές. Δηλαδή αν υποθέσουμε ότι στο παραπάνω παράδειγμα υπήρχε και μια μεταβλητή κανονική του κυρίως προγράμματος με το όνομα Month τότε μέσα στην with γράφοντας Month εννοούμε το πεδίο του aDate, και όχι την κανονική μεταβλητή. Γενικά όμως πρέπει να προσέχουμε αυτές τις περιπτώσεις και να μην χρησιμοποιούμε μεταβλητές με όνομα ίδιο με των πεδίων των records που έχουμε στο πρόγραμμά μας.

Τα αρχεία είναι ότι πιο χρήσιμο υπάρχει στον προγραμματισμό μετά τις μεταβλητές. Η βασική τους χρήση είναι η αποθήκευση από τα προγράμματα κάποιων δεδομένων έτσι ώστε να μπορούν να τα χρησιμοποιήσουν ξανά την επόμενη φορά που θα τρέξουν. Π.χ. σε ένα παιχνίδι που θέλουμε να κρατάει τα High Scores, θα πρέπει να τα γράφουμε στον υπολογιστή μόνιμα. Αυτό δεν μπορεί να γίνει παρά μόνο αν τα στοιχεία αυτά γραφούν σε σκληρό δίσκο ή δισκέτα. Ένα παιχνίδι λοιπόν, "σώζει" τα scores μέσα σε ένα αρχείο, με τέτοιο τρόπο ώστε όταν ξανατρέξουμε το παιχνίδι να μπορεί να διαβάσει τα High Scores από το αρχείο, να τα χρησιμοποιήσει και να τα ενημερώσει.

Διαχωρισμός Αρχείων από το DOS

Τα αρχεία χωρίζονται από το DOS σε δύο βασικές κατηγορίες. Τα **text files (ASCII files)** και τα **binary files**. Ουσιαστικά τα text files είναι μία ειδική περίπτωση αρχείων που αποθηκεύουμε κείμενα σε μορφή γραμμών. Κάποιοι ειδικοί χαρακτήρες (από #0 έως #31 του κώδικα ASCII) τελούν κάποια special - "ειδική" δουλειά και χρησιμοποιούνται για να δείχνουν π.χ. το τέλος της κάθε γραμμής, το τέλος του κειμένου - αρχείου κ.λ.π.

Στα binary files, δεν υπάρχει κανένας περιορισμός. Μπορούμε να γράψουμε τα πάντα σε ότι μορφή θέλουμε. Ουσιαστικά η μικρότερη πληροφορία που μπορούμε να γράψουμε σε ένα αρχείο είναι ένα byte (ή ένα char - το ίδιο είναι). Έτσι μπορούμε να πούμε ότι τα αρχεία γενικά περιέχουν χαρακτήρες. Τα binary μπορούν να έχουν οποιοδήποτε χαρακτήρα σε οποιαδήποτε θέση, ενώ στα text **κάποιοι** χαρακτήρες εξυπηρετούν κάποιους συγκεκριμένους σκοπούς (τέλος γραμμής, τέλος κειμένου).

Διαχωρισμός Αρχείων λόγω Προσπέλασης τους

Ανάλογα με τον τρόπο που διαβάζουμε ή γράφουμε τα αρχεία, τα χωρίζουμε σε δύο κατηγορίες. Τα **σειριακά (sequential)** και τα **τυχαίας προσπέλασης (random access)**. Η κάθε μία κατηγορία έχει τα προτερήματα και τα προβλήματα της. Γι' αυτό είναι υποχρέωση του προγραμματιστή να αποφασίζει ανάλογα με τις ανάγκες που έχει, ποιά από τις δύο κατηγορίες βολεύει κάθε φορά.

Σειριακά είναι τα αρχεία στα οποία γράφουμε τα δεδομένα το ένα πίσω από το άλλο και τα διαβάζουμε υποχρεωτικά με την ίδια σειρά με την οποία τα γράψαμε.

Τυχαίας Προσπέλασης είναι τα αρχεία στα οποία γράφουμε τα δεδομένα σε όποιο σημείο του αρχείου θέλουμε και διαβάζουμε από όποιο σημείο θέλουμε.

Τα Αρχεία στην Turbo Pascal

Η Turbo Pascal μας δίνει τη δυνατότητα να φτιάχνουμε και να χρησιμοποιούμε και τα δύο είδη αρχείων που υποστηρίζει το DOS, **text files** και **binary files**. Μάλιστα τα binary files που δημιουργούμε μπορεί να είναι δύο μορφών. **Typed** και **Untyped binary files**. Τα typed χρησιμεύουν όταν τα δεδομένα μας είναι ομοιογενή (π.χ. στοιχεία αθλητών μπάσκετ), ενώ τα untyped όταν τα δεδομένα μας είναι ανομοιογενή (π.χ. εικόνες) .

Σημείωση. Ο τρόπος προσπέλασης των text files είναι υποχρεωτικά σειριακός. Γι' αυτό πολλοί (λανθασμένα) λέγοντας text files εννοούν σειριακά αρχεία και αντίστροφα. Δεν υπάρχει στην πραγματικότητα τέτοια σχέση. Απλά τα text files είναι υποχρεωτικά σειριακά. Τα σειριακά αρχεία όμως δεν είναι υποχρεωτικά text files.

Text Files (Αρχεία Κειμένου)

Τα αρχεία κειμένου είναι αρχεία Εισόδου ή Εξόδου και δηλώνονται στην Turbo Pascal χρησιμοποιώντας τον τύπο δεδομένων **text**. Όταν "ανοιχθεί" ένα τέτοιο αρχείο, η γλώσσα το "βλέπει" σαν ένα σύνολο από γραμμές που όλες τελειώνουν με τους χαρακτήρες #13#10 (Carriage Return/Line Feed) που ονομάζονται και **End Of Line (EOL) Mark**.

Το EOL Mark είναι ένα σημάδι το οποίο μας δείχνει που τελειώνει μια γραμμή του αρχείου μας και που αρχίζει η επόμενη. Πολλές φορές τις γραμμές ενός text file τις ονομάζουμε εγγραφές.

Είναι προφανές ότι όπως σε ένα κείμενο κάθε γραμμή δεν έχει το ίδιο μήκος με μια άλλη, έτσι και σε ένα text file το μήκος της μιας γραμμής (εγγραφής) δεν είναι το ίδιο με το μήκος μιας άλλης. Αυτό είναι και το βασικότερο χαρακτηριστικό των αρχείων αυτών και συνεπάγεται τη σειριακή τους προσπέλαση. Σειριακή προσπέλαση σημαίνει ότι άμα θέλω να "διαβάσω" π.χ. την 5η γραμμή του κειμένου, δεν υπάρχει τρόπος να τη ζητήσω, αλλά πρέπει

να διαβάζω τις τέσσερις πρώτες, οπότε ο υπολογιστής (το DOS ουσιαστικά) να ξέρει που βρισκόμαστε και μετά να διαβάσω και την 5η που στην ουσία ζητώ.

Η μορφή ενός αρχείου κειμένου είναι κάπως έτσι :

Line 1	EOL	Line 2	EOL	Line 3	EOL	Line 4	EO	EOF
							L		
								

Στο τέλος κάθε text file υπάρχει πάντοτε ένας δείκτης που ονομάζεται **End Of File (EOF) Mark**. Ο δείκτης αυτός είναι ο χαρακτήρας **#26** από τον πίνακα ASCII που είναι γνωστός και ως **Control-Z**, γιατί μπορούμε να τον πληκτρολογήσουμε όταν βρισκόμαστε σε επίπεδο DOS πατώντας τον συνδυασμό πλήκτρων Ctrl και Z.

Μπορούμε να θεωρήσουμε ότι υπάρχει ένας δείκτης σε κάθε αρχείο ο οποίος δείχνει ποιόν χαρακτήρα θα διαβάσουμε ή θα γράψουμε με την επόμενη εντολή που θα αναφέρεται στο αρχείο. Είναι κάτι σαν τον κέρσορα που βρίσκεται στην οθόνη του υπολογιστή μας και λειτουργεί ανάλογα. Π.χ. αν ανοίξουμε ένα text file για να διαβάσουμε όλα τα στοιχεία του, ο δείκτης αυτός θα μετακινηθεί στην αρχή στον πρώτο χαρακτήρα του αρχείου. Μόλις διαβάσουμε έναν χαρακτήρα θα μετακινηθεί στον επόμενο κ.ο.κ. Όταν διαβάζουμε ολόκληρη τη γραμμή του αρχείου ο δείκτης μετακινείται στην αρχή της επόμενης γραμμής (μετά το EOL της γραμμής που βρισκόμαστε).

Χειρισμός των Text Files

Η χρησιμότητα των αρχείων, είναι γενικά η εξής : Να μπορούμε να γράφουμε δεδομένα στη δισκέττα ή στο σκληρό δίσκο (εκεί δηλαδή που παραμένουν αναλλείωτα), έτσι ώστε να μπορούμε να τα ξαναχρησιμοποιήσουμε όποτε θέλουμε, να τα αλλάξουμε, να τα σβήσουμε κ.λ.π.

Η πιο χαρακτηριστική περίπτωση είναι τα προγράμματα επεξεργασίας κειμένου που όταν γράφουμε ένα κείμενο μπορούμε να το γράψουμε σε δισκέττα έτσι ώστε να μπορούμε να το έχουμε κρατημένο για μια μελλοντική χρήση. Το κείμενό μας θα γραφτεί στη δισκέττα σε σειριακή μορφή ανεξάρτητα από το πρόγραμμα που χρησιμοποιούμε. Αυτό το καταλαβαίνουμε λόγω της ανομοιογένειας των γραμμών των κειμένων. Οι editors είναι μία κατηγορία προγραμμάτων που μας επιτρέπουν να γράφουμε απλά και μικρά text files χωρίς

ιδιαίτερα χαρακτηριστικά (όπως γίνεται στους μεγάλους επεξεργαστές κειμένου π.χ. Word for Windows).

Το λειτουργικό μας σύστημα το DOS, έχει τον δικό του editor (τον EDIT) ο οποίος δημιουργεί και γράφει αρχεία κειμένου. Επίσης ο editor της ίδιας της Turbo Pascal "σώζει" τα προγράμματά μας σ' αυτήν τη μορφή (.PAS). Ακόμα η BASIC (.BAS), η C (.C ή .CPP) κ.λ.π. "σώζουν" χρησιμοποιώντας text files.

Εμείς τώρα σαν προγραμματιστές έχουμε την ανάγκη να γράφουμε προγράμματα τα οποία είναι όσο το δυνατόν πιο χρήσιμα για αυτούς που θα τα χειριστούν. Για παράδειγμα ας υποθέσουμε ότι θέλουμε να κάνουμε ένα πρόγραμμα το οποίο να δίνει στον χρήστη τη δυνατότητα να αλλάξει τα χρώματα της οθόνης και να τα γράψει στη δισκέττα έτσι ώστε να μπορεί να τα χρησιμοποιήσει την επόμενη φορά που θα "τρέξει" το πρόγραμμά μας. Το ερώτημα είναι : Πως μπορούμε μέσα από το πρόγραμμά μας να καταχωρήσουμε αυτά τα δεδομένα (χρώματα) σε ένα αρχείο;

Οταν θέλουμε να χρησιμοποιήσουμε ένα text file στο πρόγραμμά μας πρέπει να ακολουθήσουμε τα ακόλουθα βήματα :

1. Να δηλώσουμε μια μεταβλητή τύπου text η οποία θα είναι ουσιαστικά το αρχείο μας.
2. Να δώσουμε ένα όνομα αρχείου στη μεταβλητή text, για να μπορέσει να γραφτεί στη δισκέττα, σύμφωνα με τους κανόνες του MS-DOS (όνομα αρχείου μέχρι 8 χαρακτήρες και επέκταση μέχρι 3 χαρακτήρες)
3. Να "ανοίξουμε" το αρχείο για να μπορέσουμε να διαβάσουμε ή να γράψουμε δεδομένα (ανάλογα με τη δουλειά που θέλουμε να κάνουμε, ανοίγουμε και το αρχείο με συγκεκριμένο τρόπο).
4. Αφού χρησιμοποιήσουμε το αρχείο, να το κλείσουμε.

Procedures και Functions για Text Files

- **procedure Assign (var f; Filename : string);**

f είναι μια μεταβλητή οποιουδήποτε τύπου αρχείου και Filename είναι το όνομα του αρχείου σύμφωνα με τους περιορισμούς του MS-DOS. Η procedure Assign δίνει στη μεταβλητή αρχείου f, το όνομα Filename. Π.χ.

```
Assign (MyTextFile, 'PROGRAM.BAT');
```

- **procedure Reset (var f);**

Ανοίγει το αρχείο `f`, για ανάγνωση δεδομένων. Προηγουμένως πρέπει οπωσδήποτε να έχουμε κάνει Assign στη μεταβλητή μας. Το αρχείο πρέπει να υπάρχει στο δίσκο.

- **procedure Rewrite (var f);**

Δημιουργεί και ανοίγει ένα καινούργιο αρχείο για να γράψουμε δεδομένα. Αν το αρχείο υπάρχει ήδη στη δισκέττα τότε η Rewrite θα σβήσει όλα τα περιεχόμενα του και θα ξεκινήσει πάλι να γράφει από την αρχή.

- **procedure Append (var f : text);**

Η procedure Append υπάρχει μόνο για τα text files (ενώ οι δύο προηγούμενες και για τα τυχαίας προσπέλασης) και ανοίγει ένα σειριακό αρχείο για να προσθέσει δεδομένα. Εννοείται ότι το αρχείο υπάρχει στη δισκέττα. Αλλιώς θα δημιουργηθεί ένα Run-Time Error και το πρόγραμμα θα σταματήσει.

- **procedure Close (var f);**

Κλείνει ένα ανοιχτό αρχείο και ενημερώνει τη δισκέττα ή το δίσκο με τα τελευταία δεδομένα.

- **procedure Read (var f ,);**

- **procedure ReadLn (var f ,);**

Και οι δύο procedures διαβάζουν δεδομένα από το ανοιχτό αρχείο `f`, με τη διαφορά ότι ενώ οι πρώτη διαβάζει χαρακτήρα - χαρακτήρα μέχρι να βρεί κενό, η δεύτερη διαβάζει ολόκληρη γραμμή ενός αρχείου κειμένου (και το EOL Mark αλλά δεν το επιστρέφει).

- **procedure Write (var f ,);**

- **procedure WriteLn (var f ,);**

Και οι δύο γράφουν δεδομένα στο ανοιχτό αρχείο `f`. Η πρώτη γράφει δεδομένα κανονικά όπως θα τυπωνόταν στην οθόνη, ενώ η δεύτερη αφού γράψει ότι έχουμε δώσει, θα γράψει στο αρχείο και ένα EOL Mark.

- **function EOF (var f) : boolean;**

Η συνάρτηση αυτή επιστρέφει την τιμή true αν έχουμε φτάσει στο τέλος ενός αρχείου και false αν δεν έχουμε φτάσει. Χρησιμοποιείται όταν δεν ξέρουμε πόσες εγγραφές έχουμε σε ένα αρχείο (συνήθως στα σειριακά).

Παραδείγματα με Text Files

- **Δημιουργία ενός Text File**

Να γίνει πρόγραμμα το οποίο να δημιουργεί ένα αρχείο κειμένου με το όνομα PERSONAL.TXT και να καταχωρεί το όνομα σας και τη διεύθυνσή σας.

```
var
    TheFile : text;
begin
    Assign (TheFile, 'PERSONAL.TXT');

    Rewrite (TheFile);

    Writeln (TheFile, 'Frank Borland');
    Writeln (TheFile, 'Rocky Mountains');
    Close (TheFile);
end.
```

Η μεταβλητή του αρχείου μας.

Δίνουμε στη μεταβλητή TheFile το όνομα 'PERSONAL.TXT'. Από δω και πέρα θα χρησιμοποιούμε τη μεταβλητή TheFile και θα εννοούμε αυτό το αρχείο. Δημιουργούμε το αρχείο για γράψιμο. Αν το αρχείο υπήρχε, τώρα θα έχει σβηστεί και μηδενιστεί. Γράφουμε το όνομά μας. Γράφουμε το επίθετο μας. Κλείνουμε το αρχείο και αυτό ήταν.

- **Πρόσθεση στοιχείων σε Text File**

Ν.γ.π. το οποίο να προσθέσει στο παραπάνω αρχείο κειμένου το τηλέφωνό σας, κατόπιν μια κενή γραμμή και τα στοιχεία ενός γνωστού σας.

```
var
    OneFile : text;
begin
    Assign (OneFile, 'PERSONAL.DAT');

    Append (OneFile);

    Writeln (OneFile, '(216) 543-2100');
    Writeln (OneFile);
    Writeln (OneFile, 'Philip Kahn');

    Writeln (OneFile, 'Silicon Valley');
    Writeln (OneFile, '(621) 555-6789');
    Close (OneFile);
end.
```

Η μεταβλητή δεν έχει καμία σχέση με αυτήν που χρησιμοποιήσαμε στο προηγούμενο πρόβλημα, άρα την ονομάζουμε όπως θέλουμε.

Δίνουμε όμως στη μεταβλητή το ίδιο filename με πριν (για να ξέρει πιο αρχείο της δισκέτας να ενημερώσει). Ανοίγουμε το αρχείο για να προσθέσουμε δεδομένα στο τέλος του. Γράφουμε το τηλέφωνο μας. Γράφουμε μία κενή γραμμή. Γράφουμε τα στοιχεία του γνωστού μας (όνομα)... (διεύθυνση)... (τηλέφωνο). Κλείνουμε το αρχείο.

- **Εκτύπωση δεδομένων ενός Text File**

Ν.γ.π. το οποίο να ζητάει το όνομα ενός αρχείου κειμένου, να το ανοίγει και να το τυπώνει στην οθόνη.

<pre> var AFile : text; Filename, ALine : string; begin write ('Δώσε όνομα αρχείου : '); readln (Filename); Assign (AFile,Filename); Reset (Afile); while not EOF (AFile) do begin Readln (AFile,ALine); writeln (ALine); end; Close (Afile); end. </pre>	<p>Η μεταβλητή του αρχείου. Το όνομα του αρχείου. Μία γραμμή - εγγραφή του αρχείου. Θα χρησιμοποιηθεί για να διαβάζουμε από το αρχείο.</p> <p>Ζητάμε το όνομα... ...και το καταχωρούμε στη μεταβλητή Filename. Δίνουμε το όνομα στη μεταβλητή του αρχείου μας και το ανοίγουμε για διάβασμα. Όσο δεν έχουμε βρεί το τέλος του αρχείου ακόμα.... ...διαβάζουμε μία γραμμή από το αρχείο και την τυπώνουμε στην οθόνη. Αυτό συνεχίζεται μέχρι να φτάσουμε στο τέλος του αρχείου οπότε η EOF θα μας δώσει true. Κλείνουμε το αρχείο.</p>
--	---

Προσέξτε τη χρήση της συνάρτησης EOF στο παραπάνω παράδειγμα. Ο τρόπος που τη χρησιμοποιούμε είναι συγκεκριμένος. Πάντοτε όποτε θέλουμε να διαβάσουμε τα περιεχόμενα ενός σειριακού αρχείου από την αρχή μέχρι το τέλος του, χρησιμοποιούμε τον παραπάνω τρόπο.

Binary Files (Random Access)

Τα binary (δυναδικά) αρχεία στην Turbo Pascal έχουν τη δυνατότητα να είναι τυχαίας προσπέλασης (Random Access Files - RAF). Αφού δηλαδή "ανοίξουμε" ένα τέτοιο αρχείο μπορούμε να μετακινηθούμε σε όποιο σημείο του θέλουμε και να διαβάσουμε ή να γράψουμε κάποια πληροφορία. Σε αντίθεση με τα text files που τα ανοίγουμε αποκλειστικά για διάβασμα (Reset) ή γράψιμο (Append, Rewrite) τα binary files ανοίγουν και έχουν τη δυνατότητα Read-Write εξ' ορισμού.

Τα binary files αναλόγως του τί θέλουμε να κάνουμε, είναι typed ή untyped. Σε γενικές γραμμές αυτή η διαφορά μπορεί να εξηγηθεί ως εξής :

Typed binary είναι ένα αρχείο το οποίο αποτελείται από εγγραφές που όλες έχουν του ίδιου είδους πληροφορίες, ενώ

Untyped binary είναι ένα αρχείο στην πιο απλή του μορφή, ένα "τσουβάλι" από bytes (chars άμα θέλετε) που μπορούμε να γράψουμε εντελώς ότι θέλουμε.

Θα μπορούσαμε ακόμα να πούμε ότι τα untyped binary files είναι αρχεία στην πιο low level μορφή.

Typed Binary Files

Τα typed binary αρχεία λοιπόν είναι αρχεία που αποτελούνται από όμοιες εγγραφές. Π.χ. καρτέλλες μαθητών, καρτέλλες αυτοκινήτων κ.λ.π. Χρησιμοποιούνται κυρίως για αρχειοθέτηση ομοίων πραγμάτων. Είναι βέβαια αλήθεια ότι τα τελευταία χρόνια υπάρχει μία τάση να εγκαταλείψουμε τις κατεξοχήν γλώσσες προγραμματισμού (Pascal, BASIC, C, C++) για να κάνουμε αρχειοθέτηση, και να περάσουμε στις λεγόμενες DBMS (DataBase Management Systems) και RDBMS (Relational DBMS) που μας δίνουν τεράστιες δυνατότητες αρχειοθέτησης (δίκτυα, ασύμβατοι υπολογιστές κ.λ.π.).

Τα typed binary αρχεία έχουν δυνατότητες Random Access και γι' αυτό συχνά τα αποκαλούμε και απλά Random Access Files. Αυτό επειδή ένα τέτοιο αρχείο δεν μπορεί να είναι text. Ετσι παρακάτω είτε διαβάσετε Typed Binary, είτε Random Access σημαίνει ουσιαστικά το ίδιο πράγμα.

Στα Typed Binary αρχεία το μήκος (μέγεθος) κάθε εγγραφής τους είναι ίδιο. Κάθε εγγραφή είναι του ίδιου τύπου δεδομένων. Είτε είναι απλός τύπος (integer, real), είτε πιο σύνθετος (record, array) ισχύουν τα ίδια ακριβώς πράγματα. Ακόμα δεν υπάρχει κανένας δείκτης για το τέλος της κάθε εγγραφής, ούτε για το τέλος του αρχείου όπως στα text files. Ένα RAF μπορεί να είναι κάπως έτσι :

Record 0	Record 1	Record 2	Record 3	Record 4	Record 5	

Όπως και στα text files έτσι και στα RAF όμως, υπάρχει ένας δείκτης ο οποίος μας δείχνει σε ποιο record βρισκόμαστε κάθε φορά. Ετσι αν βρισκόμαστε π.χ. στο record 2 και διαβάσουμε τα περιεχόμενα του τότε ο δείκτης αυτόματα θα μετακινηθεί στο record 3. Καταλαβαίνουμε ότι για τον υπολογιστή δεν είναι καθόλου δύσκολο να διαβάσει όποια εγγραφή του πούμε εμείς γιατί αν ξέρει το μήκος της εγγραφής μπορεί να κάνει έναν πολλαπλασιασμό και να βρεί από ποιο byte του αρχείου αρχίζει το κάθε record. Π.χ. αν έχουμε ένα RAF με μήκος record 100 bytes και πούμε στον υπολογιστή ότι θέλουμε να διαβάσουμε το record 2 τότε αυτόματα ο δείκτης θα μετακινηθεί στο 201ο byte (2 records "προσπερνάμε" (το 0 και το 1) και πάμε στην αρχή του 2).

Υπάρχει ακόμα μια ουσιαστική διαφορά ανάμεσα στα text και στα Random Access αρχεία. Όταν ανοίγουμε ένα text file δεν μπορούμε να γράψουμε και να διαβάσουμε ταυτόχρονα. Είμαστε υποχρεωμένοι να το ανοίξουμε για μια συγκεκριμένη δουλειά. Στα binary αρχεία

όμως, από τη στιγμή που θα τα ανοίξουμε μπορούμε να γράψουμε αλλά και να διαβάσουμε δεδομένα ταυτόχρονα.

Επιπλέον στα RAF, είναι δυνατόν να κάνουμε και αυτό που ονομάζεται "διόρθωση" εγγραφής, διαβάζοντας ένα record στη μνήμη, αλλάζοντας τα περιεχόμενά του και γράφοντας το πάλι στην ίδια θέση του αρχείου.

Οργάνωση και χειρισμός ενός Typed Binary File

Στην Turbo Pascal για να ανοίξουμε και να χρησιμοποιήσουμε ένα typed binary file πρέπει να χρησιμοποιήσουμε μια μεταβλητή του τύπου file of. Γράφουμε λοιπόν στις δηλώσεις var :

όνομα μεταβλητής : **file of** τύπος δεδομένων ;

Τα βήματα που πρέπει να ακολουθήσουμε, παρόμοια με τα text files, είναι :

1. Ορίζουμε μια μεταβλητή τύπου **file of** κάποιο τύπο δεδομένων. Μαζί με αυτή τη μεταβλητή θα χρειαστούμε **πάντα** και μία απλή μεταβλητή του ίδιου τύπου δεδομένων με το αρχείο.
2. Δίνουμε ένα όνομα του MS-DOS στη μεταβλητή file.
3. Ανοίγουμε το αρχείο.
4. Χρησιμοποιούμε το αρχείο (διαβάσματα - γραψίματα κ.λ.π.)
5. Κλείνουμε το αρχείο.

Τις περισσότερες φορές ο τύπος δεδομένων που χρησιμοποιούμε είναι ένα record. Αν δηλαδή θέλουμε να καταχωρήσουμε στοιχεία μαθητών σε ένα αρχείο, "φτιάχνουμε" στις δηλώσεις type έναν τύπο δεδομένων record με όνομα π.χ. OneStudent και ορίζουμε το αρχείο μας σαν file of OneStudent. Δεν πρέπει να ξεχνάμε ότι παράλληλα με τη μεταβλητή του αρχείου, χρειαζόμαστε οπωσδήποτε και μία μεταβλητή του τύπου δεδομένων που χρησιμοποιήσαμε για να μπορούμε να διαβάζουμε/γράφουμε στο αρχείο.

Procedures και Functions για Typed Binary Files

- **procedure Assign (var f; Filename : string);**

Ισχύουν τα όσα είπαμε στα σειριακά αρχεία, δηλαδή κάνουμε τη σύνδεση μεταξύ της μεταβλητής μας και ενός ονόματος αρχείου του MS-DOS.

- **procedure Rewrite (var f);**

Δημιουργεί και ανοίγει ένα καινούργιο αρχείο με το όνομα που δώσαμε στην Assign. Αν το αρχείο υπάρχει στη δισκέτα τότε θα σβηστεί και στη θέση του θα δημιουργηθεί ένα κενό αρχείο.

- **procedure Reset (var f);**

Ανοίγει ένα αρχείο τυχαίας προσπέλασης. Ο δείκτης μετακινείται στην αρχή του αρχείου (record 0)

- **procedure Seek (var f; n : longint);**

Μετακινεί τον δείκτη του αρχείου f στο record n. Το πρώτο record του αρχείου έχει αριθμό 0, δηλαδή κάνοντας Seek (MyFile,0) μετακινούμε τον δείκτη του αρχείου στο πρώτο του record (εκεί που βρίσκεται δηλαδή μετά από ένα Reset).

- **procedure Truncate (var f);**

Σβήνει όλα τα records του αρχείου από το record στο οποίο βρισκόμαστε μέχρι το τέλος του αρχείου. Χρησιμοποιείται με πολύ προσοχή!

- **procedure Write (var f; <μεταβλητή τύπου δεδομένων του αρχείου>);**

Γράφει στο αρχείο στην τρέχουσα θέση του δείκτη του αρχείου, μία εγγραφή.

- **procedure Read (var f; <μεταβλητή τύπου δεδομένων του αρχείου>);**

Διαβάζει τα περιεχόμενα της τρέχουσας εγγραφής του αρχείου και τα δίνει στη μεταβλητή που δίνουμε.

- **function FilePos (var f) : longint;**

Επιστρέφει την τρέχουσα θέση του δείκτη του αρχείου. Αν βρισκόμαστε στο πρώτο record τότε η FilePos θα επιστρέψει τον αριθμό 0. Αν η τρέχουσα θέση είναι το τέλος του αρχείου (αν EOF(f) = true) τότε η FilePos επιστρέφει το μέγεθος του αρχείου (σε records).

- **function FileSize (var f) : longint;**

Επιστρέφει το πλήθος των record του αρχείου. Αν η FileSize επιστρέψει π.χ. τον αριθμό 100, σημαίνει ότι υπάρχουν 100 records αριθμημένα από 0 - 99. Κάνοντας Seek στο record 100 σημαίνει ότι ο δείκτης του αρχείου θα μετακινηθεί μετά το τέλος του και το Eof θα γίνει true.

Παραδείγματα με Typed Binary Files

Τα παραδείγματα που ακολουθούν είναι ουσιαστικά μια ενιαία εφαρμογή αρχειοθέτησης. Αν και προσωπικά πιστεύω ότι για εφαρμογές αρχειοθέτησης δεν πρέπει πια να χρησιμοποιούμε Turbo Pascal (ή κάποια άλλη γλώσσα), αλλά RDBMS's, παραθέτω εδώ την εφαρμογή αυτή σαν ένα πολύ καλό παράδειγμα δυνατοτήτων της Pascal και των typed binary αρχείων.

Τα αρχεία τυχαίας προσπέλασης είναι συνήθως αρχεία των οποίων το μέγεθος δεν μεταβάλλεται. Χωρίς αυτό να αποτελεί κανόνα, στις περισσότερες εφαρμογές υπάρχει ένα πρόγραμμα το οποίο κάνει την εγκατάσταση των αρχείων (installation) και ένα δεύτερο πρόγραμμα (το κυρίως πρόγραμμα) το οποίο χειρίζεται το αρχείο. Είναι προφανές ότι το δεύτερο πρόγραμμα είναι και το πιο σημαντικό. Παρακάτω θα δούμε μια απλή εφαρμογή με ένα αρχείο τυχαίας προσπέλασης το οποίο θα έχει σε κάθε record τα παρακάτω πεδία :

Ονοματεπώνυμο Μαθητή	: 40 χαρακτήρες
Βαθμοί σε 4 tests	: πίνακας με 4 bytes

Όπως είναι φανερό το μέγεθος του κάθε record είναι $41+4 = 45$ bytes γιατί ένα string[40] καταλαμβάνει 41 bytes χώρο στη μνήμη.

- **Installation Αρχείου (Ανεξάρτητο πρόγραμμα)**

type

```
FileRec = record
    Name   : string[40];
    Marks  : array[1..4] of byte;
end;
```

Η δήλωση του record.

var

```
TheFile      : file of FileRec;
StudentsCard : FileRec;
```

Η μεταβλητή του αρχείου.
Η μεταβλητή που θα χρησιμοποιούμε στα διαβάσματα - γραψίματα στο αρχείο.

<pre> counter : byte; begin with StudentsCard do begin Name := ''; for counter := 1 to 4 do Marks[counter] := 0; end; Assign (TheFile, 'STUDENTS.DAT'); Rewrite (TheFile); for counter := 1 to 100 do Write (TheFile, StudentsCard); Close (TheFile); end. </pre>	<p>Ενας μετρητής.</p> <p>Μηδενίζουμε το StudentsCard. Με αυτό θα γεμίσουμε το αρχείο άδειες εγγραφές...</p> <p>Γράφω 100 φορές... ...to StudentsCard.</p>
---	---

• Πρόγραμμα διαχείρισης του αρχείου

Το πρόγραμμα διαχείρισης του αρχείου θα είναι βασισμένο οπωσδήποτε σε ένα μενού επιλογών. Οι βασικές επιλογές είναι οι εξής :

1. Εισαγωγή Στοιχείων Μαθητή
2. Διόρθωση Στοιχείων Μαθητή
3. Διαγραφή Μαθητή
4. Κατάλογος Μαθητών
5. Εξοδος από το πρόγραμμα

Οι διάφορες επιλογές του προγράμματος θα είναι γραμμένες σε procedures. Στο σημείο αυτό θα γράψουμε τις δηλώσεις του προγράμματος καθώς και το main program, ενώ παρακάτω θα ακολουθήσουν οι procedures με κάποια μικρή ανάλυση.

```

uses
  Crt;

type
  FileRecord = record
    Name   : string[40];
    Marks  : array[1..4] of byte;
  end;

var
  TheFile       : file of FileRecord;
  StudentsCard  : FileRecord;
  ProgFinished  : boolean;
  Choice        : byte;

```

{ Εδώ θα γραφούν οι procedures για κάθε επιλογή του μενού }

```

begin
  Assign (TheFile, 'STUDENTS.DAT');
  Reset (TheFile);
  ProgFinished := false;
  while not ProgFinished do
    begin
      ClrScr;

```

```

writeln (' 1. Εισαγωγή Στοιχείων Μαθητή ');
writeln (' 2. Διόρθωση Στοιχείων Μαθητή ');
writeln (' 3. Διαγραφή Μαθητή ');
writeln (' 4. Κατάλογος Μαθητών ');
writeln (' 5. Εξοδος από το πρόγραμμα ');
writeln;
write ('Επιλογή : ');
readln (Choice);
case Choice of
  1 : NewStudentsRecord;
  2 : UpdateStudentsRecord;
  3 : DeleteStudentsRecord;
  4 : CatalogOfRecords;
  5 : ProgFinished := true;
end;
end;
Close (TheFile);
end.

```

• Επιλογή 1. Εισαγωγή Στοιχείων Μαθητή

Στην επιλογή αυτή θα ζητήσουμε τον αριθμό του record στο οποίο θέλουμε να γράψουμε τα στοιχεία του μαθητή. Ο αριθμός θα μπορεί να είναι από 1..100, επειδή όμως η Pascal αριθμεί τα records από 0..99 θα γράφουμε τα στοιχεία του στο προηγούμενο record από αυτό που θα μας δώσουν. Δηλαδή θα ζητάμε έναν αριθμό από 1..100 και αφαιρώντας 1 θα παίρνουμε έναν αριθμό από 0..99 και θα γράφουμε τα στοιχεία του μαθητή σε εκείνο το record.

Γράφοντας αντί για κωδικό record τον αριθμό 0 θα τερματίζει η procedure αυτή.

```

procedure NewStudentsRecord;
var
  RecNum, counter : byte;
begin
  ClrScr;
  writeln ('Εισαγωγή Στοιχείων Μαθητή');
  writeln;
  repeat
    write ('Δώσε αριθμό record : ');
    readln (RecNum);
  until RecNum in [0..100];
  if RecNum <> 0 then
    begin
      writeln;
      with StudentsCard do
        begin
          write ('Ονοματεπώνυμο : ');
          readln (Name);
          for counter := 1 to 4 do
            begin
              write ('Test ', counter, ' : ');
              readln (Marks[counter]);
            end;
          end;
        end;
      Seek (TheFile, RecNum-1);
    end;
  end;

```

Το RecNum είναι από 0..100.
Αν δώσει 0 δεν κάνει τίποτε
αλλιώς κάνει τα παρακάτω.

Ζητάμε τα στοιχεία...

...ονοματεπώνυμο...
...και 4...

...βαθμούς.

Πηγαίνουμε στη θέση RecNum-1,
γιατί μη ξεχνάμε ότι το αρχείο
είναι αριθμημένο 0..99 (εμείς
μετράμε 1..100)

```

        Write (TheFile,StudentsCard);

    end;
end;

```

Γράφουμε στο αρχείο τα στοιχεία του μαθητή.

• Επιλογή 2. Διόρθωση Στοιχείων Μαθητή

Εδώ θα πρέπει να ζητάμε κωδικό record, να διαβάζουμε από το αρχείο τα στοιχεία του record τα οποία θα τα τυπώνουμε στην οθόνη και μετά θα ζητάμε τα καινούργια στοιχεία του μαθητή. Η δομή της procedure μοιάζει πολύ με την προηγούμενη.

```

procedure UpdateStudentsRecord;
var
    RecNum,counter : byte;
begin
    ClrScr;
    writeln ('Διόρθωση Στοιχείων Μαθητή');
    writeln;
    repeat
        write ('Δώσε αριθμό record : ');
        readln (RecNum);
    until RecNum in [0..100];
    if RecNum <> 0 then
        begin
            Seek (TheFile,RecNum-1);
            Read (TheFile,StudentsCard);
            with StudentsCard do
                begin
                    writeln ('Ονοματεπώνυμο : ',Name);
                    for counter := 1 to 4 do
                        writeln ('Test ',counter,' : ',
                            Marks[counter]);
                end;
            writeln;
            writeln ('Δώσε τα νέα στοιχεία :');
            writeln;
            with StudentsCard do
                begin
                    write ('Ονοματεπώνυμο : ');
                    readln (Name);
                    for counter := 1 to 4 do
                        begin
                            write ('Test ',counter,' : ');
                            readln (Marks[counter]);
                        end;
                    end;
                end;
            Seek (TheFile,RecNum-1);
            Write (TheFile,StudentsCard);
        end;
    end;

```

Ζητάμε τον αριθμό του record που θέλει ο χρήστης να διορθώσει.

Πηγαίνουμε στη θέση που πρέπει και διαβάζουμε τα στοιχεία που θα διορθωθούν

Τα τυπώνουμε στην οθόνη.

Ζητάμε τα νέα στοιχεία.

...ονοματεπώνυμο...

...και τους 4 βαθμούς.

Πηγαίνουμε πάλι στο αρχείο και γράφουμε τα νέα (αλλαγμένα) δεδομένα.

• Επιλογή 3. Διαγραφή Μαθητή

Στην επιλογή αυτή θα τυπώσουμε τα στοιχεία του μαθητή στην οθόνη και θα ζητήσουμε από τον χρήστη να επιβεβαιώσει την διαγραφή. Για να διαγράψουμε το record απλά θα μηδενίσουμε τα στοιχεία του.

```

procedure DeleteStudentsRecord;

```



```

var
  RecNum,counter : byte;
begin
  ClrScr;
  writeln ('Διαγραφή μαθητή');
  writeln;
  repeat
    write ('Δώσε αριθμό record : ');
    readln (RecNum);
  until RecNum in [0..100];
  if RecNum <> 0 then
  begin
    Seek (TheFile,RecNum-1);
    Read (TheFile,StudentsCard);
    with StudentsCard do
    begin
      writeln ('Όνοματεπώνυμο : ',Name);
      for counter := 1 to 4 do
        writeln ('Test ',counter,' : '
                  Marks[counter]);
    end;
    writeln;
    writeln ('Enter για διαγραφή,');
    writeln ('άλλο πλήκτρο για ακύρωση : ');
    if ReadKey = #13 then
    begin
      with StudentsCard do
      begin
        Name := '';
        for counter := 1 to 4 do
          Marks[counter] := 0;
        end;
        Seek (TheFile,RecNum-1);
        Write (TheFile,StudentsCard);
      end;
    end;
  end;
end;

```

Ζητάμε τον αριθμό του record που θέλει να διαγράψει ο χρήστης.

Πηγαίνουμε στη θέση που πρέπει και διαβάζουμε τα στοιχεία που θα διορθωθούν
Τα τυπώνουμε για να τα δει ο χρήστης...

...και περιμένουμε να πατήσει Enter αν θέλει να τα διαγράψει.

Αν πατήσει Enter :

Μηδενίζουμε το StudentsCard...

...και το γράφουμε στη θέση που θέλουμε να διαγράψουμε.

• Επιλογή 4. Κατάλογος Μαθητών

Στην επιλογή αυτή θα διαβάσουμε ένα προς ένα τα records του αρχείου και θα τυπώνουμε σε κάθε γραμμή της οθόνης τα στοιχεία εκείνων των records που έχουν Name <> ". Θεωρούμε δηλαδή ότι αν ένα record δεν έχει καταχωρημένο όνομα μαθητή είναι άδεια.

```

procedure CatalogOfRecords;
var
  RecNum,RecCount,counter : byte;
begin
  ClrScr;
  writeln ('Κατάλογος μαθητών');
  writeln;
  Seek (TheFile,0);

  for RecCount := 1 to 100 do
  begin
    Read (TheFile,StudentsCard);
    if StudentsCard.Name <> '' then
      with StudentsCard do
      begin

```

Πηγαίνουμε στην αρχή του αρχείου.
100 φορές...

...διαβάζουμε από το αρχείο το StudentsCard και αν το όνομα δεν είναι κενό...

<pre> write (RecCount:3,' ',Name,' '); for counter := 1 to 4 do write (Marks[counter]:4); writeln; end; end; writeln; writeln ('Πατήστε Enter για συνέχεια'); repeat until ReadKey = #13; end; </pre>	<p>...τυπώνουμε αύξοντα αριθμό, όνομα και τους τέσσερις βαθμούς</p> <p>Περιμένουμε να πατηθεί Enter και φεύγουμε...</p>
---	---

Το παραπάνω πρόγραμμα είναι ένα πολύ απλό και λιτό πρόγραμμα επεξεργασίας ενός αρχείου, και δεν παρέχει ουσιαστικές δυνατότητες στον χρήστη. Παρόλα αυτά είναι ένα παράδειγμα χρησιμοποίησης των procedures και functions που μας δίνει η Turbo Pascal για τον χειρισμό των Random Access Files.

Ενα τέλειο πρόγραμμα είναι πολύ πιο προσεκτικό σε πολλά σημεία. Π.χ. είναι δυνατόν να γίνει καταχώρηση ενός νέου μαθητή σε record το οποίο είναι γραμμένο ήδη; Είναι δυνατόν να διορθώσουμε ένα άγραφο record; Όλα αυτά είναι σημεία τα οποία ένα σωστό πρόγραμμα δεν θα τα άφηγε στην ελευθερία του χρήστη αλλά θα προλάμβανε πολλές τέτοιες καταστάσεις.

Γενικά ο προγραμματισμός των Random Access αρχείων είναι πολυδιάστατος και αποτελεί ολόκληρο κλάδο στην επιστήμη των υπολογιστών. Έχουν εφευρεθεί πολλοί αλγόριθμοι για την ταχύτερη και πιο αξιόπιστη αναζήτηση των πληροφοριών οι οποίοι αν υλοποιηθούν κάνουν ένα πρόγραμμα πραγματικά σημαντικό και αξιοπρόσεχτο.

Βέβαια θα πρέπει για ακόμη μια φορά να επισημάνουμε την εξέλιξη διαφόρων βάσεων δεδομένων και συστημάτων διαχείρισης τους, που πραγματικά κάνουν τον προγραμματισμό αρχείων μέσα από γλώσσες όπως η Pascal και η C να μοιάζει με το να *"ανακαλύπτουμε ξανά τον τροχό"* (Re-inventing the wheel)...

Untyped Binary Files

Τα Untyped Binary Files είναι αρχεία τυχαίας προσπέλασης κι αυτά, στα οποία όμως τα στοιχεία δεν έχουν καμία απολύτως σχέση μεταξύ τους. Χωρίζονται και αυτά σε εγγραφές, οι οποίες όμως σε αντίθεση με τα Typed Binary Files δεν έχουν κάποιο συγκεκριμένο τύπο δεδομένων.

Τα αρχεία αυτά χρησιμοποιούνται σε περιπτώσεις που θέλουμε να διαβάσουμε κάτι ή να γράψουμε κάτι από το δίσκο, χωρίς να μας ενδιαφέρει τί είναι αυτό. Για παράδειγμα σε μία περίπτωση αντιγραφής ενός αρχείου, απλά θέλουμε να πάρουμε κάποια bytes από ένα αρχείο και να τα γράψουμε σε ένα άλλο. Το τί περιέχουν πραγματικά αυτά τα bytes είναι το τελευταίο πράγμα που μας απασχολεί.

Τέλος, τα Untyped Binary Files τα χειριζόμαστε με τον "κατώτερο" τρόπο που μπορεί να υπάρξει. Δηλαδή ουσιαστικά λέμε στον υπολογιστή "πάρε τόσα bytes από το αρχείο και πηγαίνε τα στην τάδε θέση της μνήμης" ή "πάρε τόσα bytes από εκείνη τη θέση μνήμης και γράψτα στο αρχείο τάδε" κ.λ.π. Οι εντολές με τις οποίες διαβάζουμε και γράφουμε στα αρχεία αυτά δεν είναι πλέον οι Read και Write, αλλά δύο νέες εντολές η BlockRead και η BlockWrite όπως θα δούμε παρακάτω.

Χειρισμός των Untyped Binary Files

Για να δηλώσουμε ένα Untyped Binary Files χρησιμοποιούμε μία μεταβλητή του τύπου file. Γράφουμε λοιπόν στα var :

όνομα μεταβλητής : **file** ;

Τα βήματα που ακολουθούμε, είναι :

1. Ορίζουμε μια μεταβλητή τύπου **file**.
2. Δίνουμε ένα όνομα του MS-DOS στη μεταβλητή file.
3. Ανοίγουμε το αρχείο.
4. Διαβάζουμε από το αρχείο και γράφουμε σ' αυτό, μετακινώντας δεδομένα από το αρχείο στη μνήμη και αντίστροφα. Οι χώροι της μνήμης που χρησιμοποιούμε είναι ουσιαστικά μεταβλητές (χρησιμοποιούμε τα ονόματά τους) οποιουδήποτε τύπου δεδομένων θέλουμε.
5. Κλείνουμε το αρχείο.

Τις περισσότερες φορές ο τύπος δεδομένων που χρησιμοποιούμε για να πάρουμε τα στοιχεία ενός Untyped Binary File είναι ένας πίνακας από bytes ή από χαρακτήρες.

Στα αρχεία αυτά, δεν υπάρχει η έννοια της γραμμής όπως στα text ή της εγγραφής όπως στα Typed Binary. Υπάρχει η έννοια του block, όπου ένα block είναι ένα κομμάτι του αρχείου, μεγέθους κάποιων bytes, συνήθως όμως μεγέθους ενός (1) μόνο byte. Οι εντολές που χρησιμοποιούμε για να διαβάσουμε ή να γράψουμε στο αρχείο, έχουν την έννοια

"διάβασε τόσα blocks" ή "γράψε τόσα blocks". Οπου βέβαια όταν 1 block = 1 byte είναι σαν να λέμε "διάβασε τόσα bytes" ή "γράψε τόσα bytes".

Procedures και Functions για Untyped Binary Files

- **procedure Assign (var f; Filename : string);**

Ισχύουν τα όσα είπαμε στα προηγούμενα είδη αρχείων.

- **procedure Rewrite (var f : file [; BlockSize : word]);**

Δημιουργεί και ανοίγει ένα καινούργιο αρχείο με το όνομα που δώσαμε στην Assign. Αν το αρχείο υπάρχει στη δισκέτα τότε θα σβηστεί και στη θέση του θα δημιουργηθεί ένα κενό αρχείο. Η νέα παράμετρος που προαιρετικά βάζουμε είναι το μέγεθος του block σε bytes. Συνήθως βάζουμε 1. Προσοχή όμως διότι αν δεν δώσουμε καθόλου αριθμό bytes, το αρχείο δημιουργείται με το προκαθορισμένο μέγεθος block = 128 bytes!!!

- **procedure Reset (var f : file [; BlockSize : word]);**

Ανοίγει ένα αρχείο τυχαίας προσπέλασης. Ο δείκτης μετακινείται στην αρχή του αρχείου (record 0). Η παράμετρος BlockSize καθορίζει το μέγεθος του block σε bytes. Ισχύουν τα όσα είπαμε πιο πάνω στη Rewrite.

- **procedure Seek (var f; n : longint);**

Μετακινεί τον δείκτη του αρχείου f στο block n. Το πρώτο block του αρχείου έχει αριθμό 0, δηλαδή κάνοντας Seek (MyFile,0) μετακινούμε τον δείκτη του αρχείου στο πρώτο του block (εκεί που βρίσκεται δηλαδή μετά από ένα Reset).

- **procedure Truncate (var f);**

Σβήνει όλα τα περιεχόμενα του αρχείου από το σημείο στο οποίο βρισκόμαστε, μέχρι το τέλος του. Χρησιμοποιείται με πολύ προσοχή!

- **procedure BlockRead (var f : file; var Buf; Count : word [; var Result : Word]);**

Διαβάζει ένα ή περισσότερα blocks από το αρχείο και τα τοποθετεί σε μία μεταβλητή. Το f είναι μία μεταβλητή Untyped Binary File, το Buf είναι το όνομα της μεταβλητής που θα δεχθεί τα δεδομένα, το Count είναι ο αριθμός των blocks που θέλουμε να διαβάσει. Δημιουργείται ένα I/O Error αν Count * (μέγεθος block) είναι μεγαλύτερο από 64Kbytes (65,535 bytes). Επίσης η μεταβλητή Result αν τη χρησιμοποιήσουμε μας δίνει τον αριθμό των blocks που τελικά διαβάστηκαν. Αν όλα όσα ζητήσαμε διαβάστηκαν επιτυχώς, τότε θα είναι Count = Result μετά την εκτέλεση της procedure. Όταν το μέγεθος του block είναι 1 byte τότε τα Count και Result αναφέρονται σε bytes.

- **procedure BlockWrite (var f : file; var Buf; Count : word [; var Result : Word]);**

Γράφει ένα ή περισσότερα blocks από κάποια μεταβλητή σε ένα αρχείο. Τα f, Buf είναι όπως και στη BlockRead. Το Count είναι ο αριθμός των blocks που θέλουμε να γράψουμε (bytes αν το μέγεθος του block = 1 byte) και το Result επιστρέφει μετά την εκτέλεση της procedure τον αριθμό των blocks που γράφτηκαν επιτυχώς. Δημιουργείται ένα I/O Error αν Count * (μέγεθος block) είναι μεγαλύτερο από 64Kbytes (65,535 bytes).

- **function FilePos (var f) : longint;**

Επιστρέφει την τρέχουσα θέση του δείκτη του αρχείου. Αν βρισκόμαστε στο πρώτο block τότε η FilePos θα επιστρέψει τον αριθμό 0. Αν η τρέχουσα θέση είναι το τέλος του αρχείου (αν EOF(f) = true) τότε η FilePos επιστρέφει το μέγεθος του αρχείου (σε blocks). Αν βέβαια έχουμε ανοίξει το αρχείο με 1 block = 1 byte τότε όλα τα παραπάνω επιστρέφουν bytes.

- **function FileSize (var f) : longint;**

Επιστρέφει το πλήθος των blocks του αρχείου. Αν, όπως συνήθως, έχουμε ορίσει 1 block = 1 byte τότε μας δίνει το μέγεθος του αρχείου σε bytes. Κάνοντας Seek (f,FileSize(f)); ο δείκτης του αρχείου θα μετακινηθεί μετά το τέλος του και το Eof θα γίνει true.

Παραδείγματα με Untyped Binary Files

Στα παραδείγματα που ακολουθούν, χρησιμοποιούνται Untyped Binary Files που το μέγεθος του κάθε block είναι 1 byte. Αυτή είναι και η 99% χρήση των αρχείων αυτών. Τα δε προβλήματα που λύνονται είναι οι πλέον συνηθισμένες εφαρμογές των UBF. Επίσης, όταν θέλουμε να γράψουμε τα περιεχόμενα οποιασδήποτε μεταβλητής, χρησιμοποιούμε τη συνάρτηση SizeOf η οποία μας δίνει το μέγεθος σε bytes μίας μεταβλητής ή ενός τύπου δεδομένων. Έτσι είμαστε σίγουροι ότι γράφουμε ή διαβάζουμε ακριβώς τόσα bytes όσα "πιάνει" η μεταβλητή που χρησιμοποιούμε.

- **Εγγραφή Δεδομένων σε Untyped Binary File**

Εχουμε έναν πίνακα από 1000 long integers, από τους οποίους χρησιμοποιούμε μόνο κάποιους από την αρχή του πίνακα. Να γράψετε σε ένα αρχείο, τον αριθμό των ακεραίων που χρησιμοποιούμε και μετά τους ίδιους τους αριθμούς.

```
{SI-}      Κλείνουμε τον έλεγχο των I/O Errors της Pascal, αλλά είμαστε υποχρεωμένοι να
           τα ελέγχουμε εμείς.

var
  f          : file;
  A          : array[1..1000] of longint;
  Num        : integer;
  Written    : integer;

begin
  Assign (f, 'ONEFILE.DAT');

  Rewrite (f,1);

  BlockWrite (f,Num,SizeOf(Num),Written);

  if Written <> SizeOf(Num) then
  begin
    writeln ('Δεν μπορώ να γράψω στο αρχείο');
    Exit;
  end;
  BlockWrite (f,A,SizeOf(A[1])*Num,Written);
  if Written <> SizeOf(A[1])*Num then
  begin
    writeln ('Δεν μπορώ να γράψω στο αρχείο');
    Exit;
  end;
  Close (f);
  writeln ('Το αρχείο γράφτηκε επιτυχώς');
end.
```

f το αρχείο, A ο πίνακας,
Num ο αριθμός των
longint του πίνακα που
χρησιμοποιούνται,
Written το αποτέλεσμα
των BlockWrite.

Δίνουμε όνομα στο
αρχείο.
Δημιουργούμε το αρχείο
με μέγεθος block = 1byte
Γράφουμε στο αρχείο τη
μεταβλητή Num η οποία
έχει SizeOf(Num) bytes!
Αν δεν γράφτηκαν όλα
blocks (=bytes)
ζητήσαμε τότε
τυπώνουμε μήνυμα
λάθους και φεύγουμε.
Γράφουμε στο αρχείο
Num long integers από
τον πίνακα A. Αν και
πάλι δε γραφτούν όλα
ζητήσαμε τυπώνουμε
μήνυμα λάθους.

Στο παραπάνω παράδειγμα για να ξέρουμε πόσα bytes από την αρχή του πίνακα A θα γράψουμε στο αρχείο χρησιμοποιούμε τον πολλαπλασιασμό `SizeOf(A[1])*Num` όπου `SizeOf(A[1])` είναι το μέγεθος του πρώτου στοιχείου του πίνακα A. Αυτό όμως είναι και το μέγεθος **κάθε** στοιχείου του πίνακα. Num είναι ο αριθμός των στοιχείων που θα γραφτούν στο αρχείο.

- **Εκτύπωση Στοιχείων Οποιοδήποτε Αρχείου**

Να τυπωθούν όλοι οι χαρακτήρες ενός αρχείου. Όσοι από αυτούς έχουν κωδικό ASCII από #0 έως #31, στη θέση τους να τυπωθεί μία τελεία.

Εχετε ποτέ προσπαθήσει να κάνετε `type command.com` ή κάποιο άλλο αρχείο και να τυπώνει διάφορες ασυναρτησίες, να χτυπάει καμπανάκια κ.λ.π.; Αυτό γίνεται γιατί η εντολή `type` τυπώνει `text files` και όχι οποιαδήποτε αρχεία. Ετσι σε ένα αρχείο που δεν είναι `text` και υπάρχουν όλοι οι χαρακτήρες του κώδικα ASCII σε οποιαδήποτε θέση, η εντολή `type` δεν βολεύει. Η άσκηση αυτή κάνει την αντίστοιχη εκτύπωση για οποιοδήποτε αρχείο, χωρίς να τυπώνει τους `control` χαρακτήρες (0-31 ASCII) που είναι πρακτικά μη εκτυπώσιμοι. Για να διαβάσει από το αρχείο χρησιμοποιεί έναν μεγάλο πίνακα στη μνήμη μεγέθους 16KBytes (= 16384 bytes). Αν το αρχείο είναι μεγαλύτερο από 16K, συνεχίζει να διαβάζει, έως ότου διαβάσει λιγότερους από 16384 χαρακτήρες που σημαίνει ότι το αρχείο τελείωσε.

```
var
  f          : file;
  s          : string;
  Buf        : array[1..16384] of char;
  NumRead,i  : word;

begin
  write ('Δώσε όνομα αρχείου : ');
  readln (s);
  Assign (f,s);
  Reset (f,1);
  repeat
    BlockRead (f,Buf,SizeOf(Buf),NumRead);
    for i := 1 to NumRead do
      if Buf[i] in [#0..#31] then
        write ('.')
      else
        write (Buf[i]);
    until NumRead <> SizeOf(Buffer);

    Close (f);
end.
```

f είναι το αρχείο, s το όνομα του, Buf ο πίνακας που θα "μπαίνουν" τα στοιχεία που διαβάζω από το f, NumRead ο αριθμός των bytes που διαβάστηκαν.

Ζητάω όνομα αρχείου, ανοίγω το αρχείο με μέγεθος block =1 byte.

Διαβάζω από το f στον πίνακα Buf, SizeOf(Buf) bytes. Τυπώνω τους πρώτους NumRead χαρακτήρες του πίνακα Buf, δηλαδή αυτούς που διαβάστηκαν από το αρχείο. Συνεχίζω εφόσον NumRead = SizeOf(Buffer) που σημαίνει ότι έχουμε κι άλλα δεδομένα στο αρχείο. Κλείνω το αρχείο.

- **Αντιγραφή Αρχείου**

Να αντιγράψετε ένα αρχείο σε ένα άλλο.

Και εδώ θα χρησιμοποιήσουμε ένα "ενδιάμεσο" πίνακα για να διαβάζουμε από το πρώτο αρχείο και να γράψουμε στο δεύτερο. Οι πίνακες αυτοί λέγονται *buffers* επειδή εξυπηρετούν μία ενδιάμεση και προσωρινή κατάσταση. Όσο πιο μεγάλος ο πίνακας *buffer* τόσο λιγότερες φορές θα διαβάσει και θα γράψει στα αρχεία το πρόγραμμα, και τόσο γρηγορότερο θα γίνει. Στα αρχεία μας ενδιαφέρει να γράφουμε όσο το δυνατόν πιο πολλά δεδομένα μαζί. Στο παράδειγμα αυτό θα χρησιμοποιήσουμε έναν πίνακα από 48K (49,152 bytes).

```
var
  f1,f2      : file;
  s1,s2      : string;
  Buf        : array[1..49152] of byte;
  NumRead    : integer;
begin
  write ('Όνομα πρωτοτύπου αρχείου : ');
  readln (s1);
  write ('Όνομα αντιγράφου αρχείου : ');
  readln (s2);
  Assign (f1,s1);
  Assign (f2,s2);
  Reset (f1,1);
  Rewrite (f2,1);
  repeat
    BlockRead (f1,Buf,SizeOf(Buf),NumRead);
    BlockWrite (f2,Buf,SizeOf(Buf),NumRead);
  until NumRead <> SizeOf(Buf);

  Close (f1);
  Close (f2);
end.
```

f1 και f2 τα δύο αρχεία, s1 και s2 τα ονόματά τους αντίστοιχα, Buf ο πίνακας, NumRead τα bytes που διαβάστηκαν.

Ζητάω τα ονόματα των αρχείων...

... και τα ανοίγω. Το πρωτότυπο με Reset, το αντίγραφο με Rewrite.

Ζητάω SizeOf(Buf) bytes από το f1 και γράφω όσα διάβασα στο f2. Αυτό συνεχίζεται μέχρι να μου δώσει λιγότερα απ' όσα ζήτησα. Κλείνω τα αρχεία.

'Έλεγχος I/O Errors

Η Turbo Pascal κάνει από μόνη της έλεγχο για λάθη, όταν χρησιμοποιούμε I/O εντολές. Σε περίπτωση που συμβεί ένα τέτοιο λάθος (πάμε να ανοίξουμε ένα αρχείο που δεν υπάρχει, πάμε να γράψουμε σε προστατευμένη δισκέτα, δεν υπάρχει αρκετός χώρος στο δίσκο, ή τη δισκέτα κ.λ.π.) η Pascal δημιουργεί ένα Run-Time Error και διακόπτει τη λειτουργία του προγράμματος.

Αν εμείς θέλουμε μπορούμε να κάνουμε μόνοι μας τον έλεγχο αυτό, μπορούμε να βάλουμε το compiler directive `{I-}` που είναι μία οδηγία προς τον compiler να σταματήσει να παράγει κώδικα για έλεγχο λαθών I/O. Αντίστοιχα υπάρχει και το `{I+}` για να ξεκινήσει και πάλι να κάνει έλεγχο λαθών.

Μετά από κάθε I/O λειτουργία, η Turbo Pascal επιστρέφει στη μεταβλητή `IOResult` τον κωδικό του λάθους που συνέβει ή 0 αν δεν συνέβει τίποτα (όλα πήγαν καλά).

Τα συνηθέστερα λάθη που αναφέρονται στην IOResult είναι :

2 File not found

Δημιουργείται από τις Reset, Append, Rename, Rewrite ή Erase όταν το όνομα του αρχείου που έχουμε κάνει Assign δεν υπάρχει.

100 Disk Read Error

Όταν πάμε να διαβάσουμε μετά το τέλος ενός αρχείου.

101 Disk Write Error

Όταν γεμίζει ο δίσκος στον οποίο γράφουμε.

102 File not assigned

Δεν έχουμε κάνει Assign σε μία μεταβλητή file και πάμε να κάνουμε Reset, Rewrite, Append, Rename ή Erase.

103 File not open

Δημιουργείται από τις Close, Read, Write, Seek, Eof, FilePos, FileSize, Flush, BlockRead, BlockWrite όταν δεν έχουμε ανοίξει το αρχείο.

104 File not open for input

Δημιουργείται από τις Read, Readln, Eof, Eoln, SeekEof, SeekEoln σε ένα text file, όταν δεν έχουμε ανοίξει το αρχείο για να διαβάσουμε (με Reset).

105 File not open for output

Δημιουργείται από τις Write, Writeln στα text files, όταν δεν έχουμε ανοίξει το αρχείο για να γράψουμε (Rewrite, Append).

106 Invalid numeric format

Δημιουργείται από τις Read, Readln αν πάμε να διαβάσουμε την τιμή μιας αριθμητικής μεταβλητής και συναντήσουμε λανθασμένους χαρακτήρες.

Δυναμικές μεταβλητές - Pointers

Ενας pointer είναι μία διεύθυνση σε κάποια δεδομένα ή σε κάποιον κώδικα του προγράμματός μας. Η χρήση των pointers γίνεται για δύο βασικούς λόγους. Πρώτον για να έχουμε πρόσβαση σε πολύ μεγαλύτερη μνήμη για μεταβλητές απ' αυτήν που μας δίνει εξ' ορισμού η Pascal (64K) και δεύτερον για να φτιάξουμε νέες πιο ευέλικτες δομές δεδομένων όπως είναι οι λίστες, τα δέντρα κ.λ.π.

Γιατί Pointers;

Αργά ή γρήγορα, κάθε προγραμματιστής σε Pascal φτάνει σε κάποια κατάσταση που χρειάζεται pointers. Είτε επειδή το πρόγραμμά του πρέπει να χειριστεί μεγάλη ποσότητα δεδομένων, είτε γιατί χειρίζεται δεδομένα αγνώστου μεγέθους που αλλάζουν κατά τη διάρκεια του προγράμματος, είτε επειδή το πρόγραμμα χρησιμοποιεί προσωρινές buffers, είτε γιατί χρειάζεται δυναμικές δομές δεδομένων που το μέγεθος τους μεγαλώνει και μικραίνει ανάλογα με το πόσα στοιχεία έχουν (λίστες, δέντρα κ.λ.π.).

Τεράστιες Ποσότητες Δεδομένων

Η Pascal μας δίνει ένα κομμάτι της μνήμης που λέγεται Data Segment για αποθήκευση των Global μεταβλητών και σταθερών των προγραμμάτων μας. Το μέγεθος του Data Segment δεν μπορεί να ξεπερνά τα 64K. Για να μπορέσουμε να το ξεπεράσουμε χρησιμοποιούμε pointers. Για παράδειγμα υποθέστε ότι θέλετε να κάνετε ένα πρόγραμμα που να χρησιμοποιεί έναν πίνακα από 400 strings με 100 χαρακτήρες maximum το καθένα. Ο πίνακας αυτός θα έχει μέγεθος περίπου 40K, το οποίο είναι λιγότερο βέβαια από 64K που έχουμε όριο. Αν τα υπόλοιπα 24K μας αρκούν για όλες τις υπόλοιπες μεταβλητές του προγράμματός μας, έχει καλώς. Αν όχι όμως, ή αν θέλουμε να έχουμε δύο τέτοιους πίνακες, τι πρέπει να κάνουμε; Η λύση είναι μόνο οι δυναμικές μεταβλητές οι οποίες πιάνουν χώρο έξω από τον Data Segment.

Δεδομένα με 'Αγνωστο Μέγεθος

Η Pascal μας υποχρεώνει να δηλώσουμε το μέγεθος όλων των πινάκων, strings που χρησιμοποιούμε στην αρχή του προγράμματος μας. Αυτό γίνεται όχι για να μας κάνει τη ζωή δύσκολη, αλλά γιατί ο compiler πρέπει να ξέρει πόσο μεγάλο Data Segment χρειάζεται το πρόγραμμά μας, έτσι ώστε να μπορεί να ζητήσει από το Dos τη μνήμη αυτή όταν το πρόγραμμά μας τρέξει. Ομως πολλές φορές θέλουμε να χρησιμοποιήσουμε πίνακες που το μέγεθος τους μπορεί να αλλάξει.

Σκεφτείτε π.χ. ένα πρόγραμμα το οποίο φορτώνει από το δίσκο ένα αρχείο κειμένου. Μπορούμε να κάνουμε έναν πίνακα από strings και σε κάθε θέση του πίνακα να βάζουμε και μία γραμμή του αρχείου. Ομως πόσες γραμμές θα μπορέσουμε να φορτώσουμε μέσα σε 64K μνήμης; Τα strings πόσο μεγάλα πρέπει να είναι; string[80] ή σκέτο string; Και καλά όταν κάποια γραμμή του κειμένου έχει πολλούς χαρακτήρες. Όταν έχουμε όμως μία κενή γραμμή γιατί αυτή να πιάνει τον ίδιο χώρο στη μνήμη με μία πλήρη;

Μπορεί το παράδειγμα να μην είναι και τόσο χαρακτηριστικό, αλλά η καλή χρήση των pointers είναι αυτή που θα ξεχωρίσει δύο φαινομενικά ίδια προγράμματα και θα κάνει τη διαφορά μεταξύ του "τρέχω επιτυχώς" ή "δεν έχω αρκετή ελεύθερη μνήμη".

Δηλώσεις και Είδη Pointers

Οι pointers δεν πρέπει να ξεχνάμε ότι είναι μεταβλητές. Είναι μεταβλητές που "δείχνουν" σε κάποια θέση της μνήμης. Υπάρχουν δύο είδη pointers. Αυτοί που "δείχνουν" σε κάποια θέση μνήμης όπου βρίσκεται κάτι που ο τύπος δεδομένων του είναι γνωστός (π.χ. δείχνουν σε ένα string) και αυτοί που δείχνουν εκεί που βρίσκεται κάτι που δεν μας ενδιαφέρει τι ακριβώς είναι. Οι πρώτοι λέγονται **typed pointers** και οι δεύτεροι **untyped pointers**.

Για να δηλώσουμε έναν typed pointer γράφουμε :

όνομα μεταβλητής : ^τύπος δεδομένων ;

και για να δηλώσουμε έναν untyped pointer :

όνομα μεταβλητής : **pointer** ;

όπου ^ είναι ο χαρακτήρας caret.

Προσέξτε το παρακάτω παράδειγμα :

var

```

p1 : ^integer;
p2 : pointer;
x : integer;

begin
  x := 19;
  p2 := @x;
  p2 := Addr(x);
  p1 := p2;
  p1^ := 21;
  writeln (x);
end.

```

Θα περιμέναμε το παραπάνω πρόγραμμα να τυπώσει την τιμή 19. Όμως αυτό θα τυπώσει 21 !!! Ας δούμε λίγο τι σημαίνουν οι παραπάνω εντολές. Το `p2 := @x` δίνει στον pointer `p2` τη διεύθυνση της μεταβλητής `x`. Έτσι τώρα μπορούμε να λέμε ότι "ο `p2` δείχνει τη μεταβλητή `x`". Ο `p2` είναι untyped pointer και μπορεί να δείξει γενικά παντού! Στην επόμενη πρόταση `p2 := Addr(x)` κάνουμε πάλι ακριβώς το ίδιο. Απλά με διαφορετικό τρόπο. Η πρόταση `p1 := p2` δίνει στον pointer `p1` την τιμή του pointer `p2`, δηλαδή τη διεύθυνση που δείχνει ο `p2`. Δηλαδή τελικά και ο `p1` παίρνει τη διεύθυνση που βρίσκεται η μεταβλητή `x`, άρα ΚΑΙ ο `p1` "δείχνει το `x`". Γενικά όταν εξισώνουμε δύο pointers είναι σαν να τους βάζουμε να δείχνουν στο ίδιο σημείο.

Λέγοντας `p1^ := 21` δίνουμε στο περιεχόμενο του `p1` (στη θέση που δείχνει δηλαδή) την τιμή 21. Προσέξτε ότι ο compiler ξέρει ότι ο `p1` δείχνει σε integers και έτσι επιτρέπει την προηγούμενη εντολή. Δεν θα επέτρεπε ποτέ να γίνει το ίδιο με τον `p2`, παρόλο που ο `p2` δείχνει και αυτός στο ίδιο σημείο. Προσέξτε επίσης, ότι το caret (^), μπαίνει αμέσως μετά το όνομα `p1`, δηλώνοντας το "σημείο που δείχνει ο `p1`". Έτσι λοιπόν σκέτο `p1` είναι ένας pointer που δείχνει σε ακέραιο, ενώ `p1^` είναι ο ίδιος ο ακέραιος! (Όπως `A` μπορεί να είναι ένας πίνακας και `A[1]` είναι ένα στοιχείο του).

Δυναμικές Μεταβλητές

Οι δυναμικές μεταβλητές είναι μεταβλητές που δεν βρίσκονται στη μνήμη σ' όλη τη διάρκεια εκτέλεσης του προγράμματος, αλλά μόνο σε συγκεκριμένες στιγμές. Όταν τις χρειαζόμαστε. Σε αντίθεση με τις κανονικές μεταβλητές (static) που σε όλη τη διάρκεια της εκτέλεσης του προγράμματός μας "πιάνουν" κάποιο χώρο στη μνήμη, οι δυναμικές δεν το κάνουν αυτό. Για να μπορέσουμε να τις χρησιμοποιήσουμε πρέπει εμείς μόνοι μας να κατακρατήσουμε (allocation) μνήμη από το Heap και όταν πια δεν τις χρειαζόμαστε να την αφήσουμε (deallocation). Το Heap είναι ένα μεγάλο μέρος της μνήμης του υπολογιστή μας το οποίο πρακτικά μένει άχρηστο στα περισσότερα προγράμματα.

Η Turbo Pascal μας δίνει δύο ζευγάρια από ρουτίνες για να κάνουμε allocation - deallocation μνήμης. Είναι οι New - Dispose και οι GetMem - FreeMem. Οι πρώτες είναι και οι πιο συνηθισμένες. Χρησιμοποιούνται στους typed pointers. Οι δεύτερες είναι λίγο πιο σπάνιες και χρησιμοποιούνται σε ένα "κατώτερο επίπεδο" με τους untyped pointers.

Επίσης για να ελέγχουμε κάθε φορά πόση μνήμη έχουμε διαθέσιμη στο Heap, υπάρχουν δύο functions, η MemAvail και η MaxAvail.

Οι δυναμικές μεταβλητές τέλος χρησιμοποιούνται περισσότερο για να δημιουργήσουμε και να χειριστούμε τις λεγόμενες δυναμικές δομές δεδομένων (λίστες, στοίβες, ουρές, δέντρα κ.λ.π.). Οι δυναμικές δομές, είναι ουσιαστικά σύνθετες δομές όπου καταχωρούνται διάφορα δεδομένα (data) και κύριο χαρακτηριστικό τους είναι ότι καταλαμβάνουν στη μνήμη τόσο χώρο, όσο ακριβώς τους χρειάζεται σε κάθε συγκεκριμένη στιγμή. Σε αντίθεση με τους πίνακες π.χ. είναι πολύ πιο ευέλικτες σ' αυτόν τον τομέα. Υστερούν βέβαια σε άλλους τομείς, όπως είναι η προσπέλαση τους. Θεωρείται πέρα από τους σκοπούς του βιβλίου αυτού η εκμάθησή τους, γι' αυτό και στεκόμαστε μόνο σε αυτήν την απλή αναφορά.

Procedures και Functions για Pointers

- **function MemAvail : longint;**

Επιστρέφει το μέγεθος της συνολικής ελεύθερης μνήμης του Heap σε bytes.

- **function MaxAvail : longint;**

Επιστρέφει το μέγεθος του μεγαλύτερου ελεύθερου μέρους της μνήμης του Heap σε bytes. Πολλές φορές το Heap, κάνοντας συνεχώς allocation - deallocation από pointers μπορεί να χωριστεί σε κομμάτια ελεύθερου χώρου. Έτσι μπορεί να έχουμε π.χ. 200K ελεύθερα, αλλά το μεγαλύτερο συνεχόμενο κομμάτι να είναι 40K. Πριν κάνουμε allocation θα πρέπει γενικά να ελέγχουμε την τιμή της MaxAvail και όχι της MemAvail, διότι το allocation πρέπει να "πάρει" ένα συνεχόμενο κομμάτι της μνήμης.

- **procedure New (var p : pointer);**

Κατακρατεί κάποια μνήμη από το Heap. Το p είναι μία μεταβλητή οποιουδήποτε είδους pointer. Συνήθως χρησιμοποιείται για **typed pointers**. Το μέγεθος της μνήμης που θα κατακρατηθεί εξαρτάται από το είδος του τύπου δεδομένων που δείχνει ο p. Μετά το allocation μπορούμε να χρησιμοποιήσουμε τη μεταβλητή που δείχνει ο p, σαν p[^]. Αν δεν υπάρχει αρκετός συνεχόμενος ελεύθερος χώρος στο Heap, θα συμβεί ένα Run-Time Error.

- **procedure GetMem (var p : pointer; Size : word);**

Κατακρατεί κάποια μνήμη από το Heap μεγέθους Size. Η διαφορά με τη New είναι ότι εδώ δίνουμε το ποσό της μνήμης (σε bytes) που θέλουμε να κατακρατήσουμε. Συνήθως χρησιμοποιείται για **untyped pointers**.

- **procedure Dispose (var p : pointer);**

Το αντίθετο της New. Αφήνει ελεύθερη τη μνήμη που κατακρατήσαμε για τον pointer p. Το ποσό της μνήμης που θα αφαιρεθεί, εξαρτάται από τον τύπο δεδομένων που δείχνει ο p. Συνήθως χρησιμοποιείται για **typed pointers**. Μετά την κλήση της Dispose ο p παύει να υπάρχει και κάθε αναφορά στο p[^] δημιουργεί σοβαρό λάθος.

- **procedure FreeMem (var p : pointer; Size : word);**

Το αντίθετο της GetMem. Αφήνει ελεύθερα Size bytes από τη μνήμη που κρατήθηκε για τον pointer p με την GetMem. Συνήθως χρησιμοποιείται σε **untyped pointers**. Είναι δική μας υποχρέωση να ελευθερώσουμε ακριβώς το ίδιο ποσό μνήμης, με αυτό που κατακρατήθηκε με την GetMem.

- **function SizeOf (x) : word;**

Το x είναι είτε μία μεταβλητή, είτε κάποιος τύπος δεδομένων. Επιστρέφει τον αριθμό των bytes που καταλαμβάνει στη μνήμη το x. Π.χ. SizeOf(integer) είναι πάντα 2. Ιδιαίτερα χρήσιμη όταν χρησιμοποιούμε FillChar, Move ή GetMem.

Η τιμή nil

Όπως όλες οι μεταβλητές της Pascal, έτσι και οι pointers όταν ξεκινάει το πρόγραμμα έχουν τυχαίες τιμές. Ένας pointer όμως, επειδή "δείχνει" σε κάποια διεύθυνση μνήμης, όπως καταλαβαίνουμε μπορεί να δείχνει παντού. Το να πάρουμε λοιπόν κάποια τιμή από έναν λάθος pointer σημαίνει ότι πιθανότατα θα πάρουμε τυχαία bits, ενώ το να δώσουμε μία τιμή

σ' αυτόν σημαίνει ότι πιθανότατα θα πάμε να γράψουμε σε κάποια άγνωστη θέση μνήμης, προκαλώντας ότι προβλήματα μπορείτε να φανταστείτε (τα συνηθισμένα "κολλήματα" του υπολογιστή). Ετσι όταν χρησιμοποιούμε pointer variables πρέπει να είμαστε ιδιαίτερα προσεκτικοί, να κάνουμε allocation πριν τους χρησιμοποιήσουμε ή να τους δίνουμε κατάλληλες διευθύνσεις μνήμης που ξέρουμε τί υπάρχει εκεί.

Η Pascal έχει μία λέξη για να δείχνει ότι κάποιος pointer δεν έχει οριστεί. Είναι η τιμή **nil** (τίποτα). Ετσι μπορούμε λοιπόν σε κάθε μεταβλητή τύπου pointer να δώσουμε την τιμή nil, και αργότερα στο πρόγραμμα να ελέγξουμε αν κάποιος pointer είναι nil, που σημαίνει ότι είναι μη ορισμένος.

Παραδείγματα με Pointers

Παρακάτω θα δούμε παραδείγματα χρήσης pointers.

- **Ελεύθερη μνήμη του Heap**

Ελέγξτε το παρακάτω πρόγραμμα και δείτε τα αποτελέσματα που τυπώνει. Στην αρχή και το τέλος του προγράμματος τα νούμερα των MemAvail και MaxAvail πρέπει να είναι ίδια.

```
var
  A,B,C : ^string;
begin
  writeln (MemAvail,' - ',MaxAvail);

  New (A);
  New (B);
  New (C);
  writeln (MemAvail,' - ',MaxAvail);
  Dispose (A);
  writeln (MemAvail,' - ',MaxAvail);

  Dispose (B);
  Dispose (C);
  writeln (MemAvail,' - ',MaxAvail);
end.
```

Δηλώνω τρεις pointers σε strings (256 bytes το καθένα).
Τυπώνω την ελεύθερη μνήμη και το μέγιστο ελεύθερο κομμάτι της. Πρέπει να είναι ίσα.
Κάνω allocation και τους τρεις pointers. Τα νούμερα πρέπει να είναι πάλι ίσα αλλά μικρότερα από τα προηγούμενα.

Deallocation του A. Τα νούμερα αλλάζουν. Τώρα η ελεύθερη μνήμη μεγάλωσε αλλά το max ελεύθερο block παραμένει ίδιο.
Αφήνω και την υπόλοιπη μνήμη. Τα νούμερα θα είναι και πάλι ίσα και μάλιστα ίσα και με τα πρώτα που τύπωσε το πρόγραμμα.

- **Τεράστιοι πίνακες**

Να γίνει η δήλωση, το allocation, η καταχώρηση κάποιας τιμής και το deallocation των μεταβλητών που χρειάζονται για να κρατήσουμε 1000 strings των 255 χαρακτήρων στη μνήμη.

Επειδή $1000 \cdot 256 =$ περίπου 256K είναι σίγουρο ότι δεν μπορούμε να φτιάξουμε έναν πίνακα στο Data Segment. Ακόμα το μεγαλύτερο μέρος μνήμης που μπορεί να γίνει allocation με μία κίνηση είναι και πάλι 64K (!). Έτσι επειδή δεν μπορούμε να έχουμε έναν pointer για έναν πίνακα των 1000 strings, μπορούμε να φτιάξουμε 1000 pointers για ένα string!

```
var
  A : array[1..1000] of ^string;
  i : integer;

begin
  for i := 1 to 1000 do
    New (A[i]);
  for i := 1 to 1000 do
    A[i]^ := 'aaa';
  for i := 1 to 1000 do
    Dispose (A[i]);
end.
```

Δηλώνω τον A σαν έναν πίνακα με 1000 στοιχεία, που το καθένα είναι ένας pointer σε string. Ο πίνακας A πιάνει μόλις $4 \cdot 1000 = 4000$ bytes στο Data Segment!!!
Κάνω allocation και τους 1000 pointers.

Θέτω σε κάθε string την τιμή 'aaa'.

Κάνω deallocation τους 1000 pointers.

Στο κεφάλαιο αυτό θα δούμε τι είναι ένα unit, πως το χρησιμοποιούμε και πως μπορούμε να φτιάξουμε δικά μας.

Τι είναι Unit

Η Turbo Pascal μας δίνει τη δυνατότητα να χρησιμοποιήσουμε πάρα πολλές έτοιμες constants, μεταβλητές, τύπους δεδομένων, procedures και functions. Επειδή είναι πραγματικά αναρίθμητες, έχουν τοποθετηθεί σε ενότητες οι οποίες ονομάζονται units. Ετσι λοιπόν οι εντολές, σταθερές, μεταβλητές κ.λ.π. που χρειάζονται για την οθόνη κειμένου υπάρχουν στο unit Crt, οι εντολές που χρειάζονται για χειρισμό διαφόρων λειτουργιών του Dos υπάρχουν στο unit Dos, οι ρουτίνες που χρειάζονται για να κάνουμε γραφικά στο unit Graph κ.λ.π.

Εκτός όμως από το να χρησιμοποιούμε έτοιμα units, που έχουμε δει στο κεφάλαιο 2 (Declarations - Δηλώσεις) η Turbo Pascal μας δίνει τη δυνατότητα να φτιάξουμε και δικά μας. Φτιάχνοντας δικά μας units μπορούμε να "σπάσουμε" μεγάλα προγράμματα σε πολλά κομμάτια τα οποία κάνουμε compilation ξεχωριστά. Ακόμα, μπορούμε να φτιάξουμε κάποια units με διάφορες χρήσιμες ρουτίνες, τις οποίες να χρησιμοποιούμε σε πολλά από τα προγράμματά μας.

Ενα unit μοιάζει πάρα πολύ με ένα κανονικό πρόγραμμα της Pascal. Το γράφουμε σε ένα ξεχωριστό αρχείο .PAS στον editor, και όταν το κάνουμε compilation δημιουργείται ένα αντίστοιχο .TPU αρχείο. Ενα ακόμα πλεονέκτημα των units είναι ότι μπορούμε αν θέλουμε να δώσουμε σε κάποιον φίλο μας μόνο το .TPU αρχείο, ή και να το πουλήσουμε ακόμα, χωρίς να του δώσουμε ουσιαστικά τον κώδικά μας (το πώς το έχουμε γράψει δηλαδή, το PAS αρχείο), δίνοντάς του όμως τη δυνατότητα να χρησιμοποιήσει ότι εμείς έχουμε φτιάξει.

Τέλος πρέπει να σημειωθεί ότι επειδή ακριβώς τα units μεταφράζονται από τον compiler ξεχωριστά και βρίσκονται σε μορφή .TPU στο δίσκο μας, δε χρειάζεται κάθε φορά να κάνουμε ξανά compilation στα units του προγράμματος μας, αλλά μόνο όταν κάνουμε μέσα σ' αυτά κάποιες αλλαγές.

Η Δομή ενός Unit

Η δομή των units μοιάζει πάρα πολύ με αυτή των προγραμμάτων :

```
unit <όνομα>;

interface

uses <λίστα με units>;
<Δηλώσεις οι οποίες φαίνονται στον "έξω" κόσμο>
<επικεφαλίδες από procedures & functions>

implementation

uses <λίστα με units>;
<Δηλώσεις οι οποίες κρατούνται "μυστικές">
<procedures και functions>

begin
<αρχικός κώδικας του unit>
end.
```

Ενα unit μπορεί να χρησιμοποιεί και άλλα units. Αυτά γράφονται σε μία δήλωση uses αμέσως μετά τη λέξη interface ή αμέσως μετά τη λέξη implementation ή και στα δύο αυτά σημεία. Η ουσία είναι ότι αν στις δηλώσεις που κάνουμε στο κομμάτι interface, χρησιμοποιούμε κάποιον τύπο δεδομένων ή σταθερά που υπάρχει σε άλλο unit, τότε για να περάσει από εκεί επιτυχώς ο compiler πρέπει να βάλουμε αυτό το unit στα uses μετά το interface.

Επικεφαλίδα ενός Unit

Ενα unit ξεκινά με την επικεφαλίδα του (υποχρεωτική). Στην επικεφαλίδα έχει ένα όνομα, του οποίου οι πρώτοι 8 χαρακτήρες πρέπει να χρησιμοποιηθούν για να σώσουμε το αρχείο. Δηλαδή αν θέλουμε να ονομάσουμε το unit μας MyTools πρέπει να το σώσουμε στο αρχείο MYTOOLS.PAS. Κάτι τέτοιο μπορούμε αν θέλουμε να το αποφύγουμε, αλλά τότε κάθε φορά που θέλουμε να χρησιμοποιούμε το unit μας, θα πρέπει να λέμε και από ποιο αρχείο θα το πάρει ο compiler, που είναι αρκετά κουραστικό.

Interface

Το interface κομμάτι ενός unit, είναι το "κοινοποιήσιμο" κομμάτι του (public). Οτι γράφουμε εκεί, "φαίνεται" από οποιονδήποτε (πρόγραμμα ή άλλο unit) χρησιμοποιήσει το unit μας. Στο κομμάτι interface κάνουμε δηλώσεις σταθερών, typed constants, μεταβλητών και τύπων δεδομένων. Επίσης γράφουμε τις επικεφαλίδες από τις procedures και τις functions που θέλουμε να είναι "ορατές" στους χρήστες του unit μας. Οι πραγματικές procedures και functions θα γραφτούν στο επόμενο κομμάτι του unit μας, το implementation.

Implementation

Το implementation (σημαίνει πραγματοποίηση) μέρος του unit είναι το "κρυφό" μέρος του. Οτι έχει δηλωθεί στο interface κομμάτι είναι "ορατό" και εδώ. Το implementation έχει και αυτό δηλώσεις από constants, μεταβλητές κ.λ.π. οι οποίες όμως δεν φαίνονται σ' αυτούς που χρησιμοποιούν το unit μας. Αυτό το κάνουμε όχι γιατί μας αρέσει να κρατάμε μυστικά (κάθε άλλο), αλλά απλά γιατί κατά τη γνώμη μας δε χρειάζεται να "δώσουμε" παραέξω κάποια πράγματα που τα χρειαζόμαστε μόνο μέσα στο unit.

Η έννοια του δομημένου προγραμματισμού είναι να αποκρύψουμε όσο το δυνατόν περισσότερα πράγματα από το υπόλοιπο κομμάτι των προγραμμάτων μας. Ετσι λοιπόν αν θέλουμε να φτιάξουμε μία ρουτίνα, πολύ όμορφη, πολύ χρήσιμη κ.λ.π. και θέλουμε να τη μοιραστούμε με τον "υπόλοιπο κόσμο", θα γράψουμε την επικεφαλίδα της στο interface κομμάτι ενός unit. Αν τώρα, η ρουτίνα μας αυτή χρησιμοποιεί, 10 μεταβλητές, 5 - 6 σταθερές, άλλες δύο μικρές και βοηθητικές ρουτίνες, δεν υπάρχει λόγος να τις βγάλουμε κι αυτές προς το έξω. Τις γράφουμε λοιπόν μέσα στο implementation κομμάτι του unit.

Οι procedures και οι functions που γράφουμε στο implementation ενός unit και έχουν δηλωθεί στο interface, μπορούν να γραφτούν και χωρίς την πλήρη επικεφαλίδα τους. Αυτή υπάρχει γραμμένη έτσι κι αλλιώς στο interface. Επειδή όμως πολλές φορές ξεχνάμε τις παραμέτρους που παίρνει μία ρουτίνα, ακόμα και την ώρα που τη γράφουμε, καλό είναι να υπάρχει και η επικεφαλίδα της, πλήρης ΚΑΙ στο implementation. Σε περίπτωση όμως που κάνουμε αλλαγή σε μία από τις δύο επικεφαλίδες ο compiler μας ειδοποιεί και μας λέει ότι έχουμε δύο ρουτίνες με το ίδιο όνομα και με διαφορετική επικεφαλίδα.

Ακόμα, όταν έχουμε δηλώσει μία ρουτίνα στο interface και δεν την έχουμε γράψει στο implementation, ο compiler παράγει ένα ακόμη λάθος.

Κύριο Μέρος - Αρχικός Κώδικας

Υπάρχει κάποιες φορές η ανάγκη προκειμένου να δουλέψουν κάποιες ρουτίνες ενός unit να γίνουν κάποια πράγματα στην αρχή - αρχή του προγράμματος (π.χ. να πάρουμε την ημερομηνία από το σύστημα). Τέτοιου είδους εντολές τις γράφουμε στο initialization section ενός unit. Αυτό το κομμάτι ξεκινά με ένα begin και τελειώνει με ένα end. (με τελεία) όπως ακριβώς και το κύριο μέρος ενός προγράμματος.

Σημειώνουμε εδώ ότι σε περίπτωση που δεν θέλουμε να έχουμε initialization section στα unit μας, δεν γράφουμε καθόλου το begin. Έτσι μετά και το τέλος της τελευταίας procedure ή function του implementation section γράφουμε απλά end.

Παράδειγμα Unit

Ας δούμε ένα unit το οποίο περιέχει τη function Maximum που επιστρέφει τον μεγαλύτερο από δύο ακεραίους.

```
unit IntMath;

interface

function Maximum ( a,b : integer ) : integer;

implementation

function Maximum ( a,b : integer ) : integer;
begin
  if a > b then
    Maximum := a
  else
    Maximum := b;
end;

end.
```

Και ας δούμε και ένα πρόγραμμα που χρησιμοποιεί αυτό το unit.

```
uses
  IntMath;
var
  x,y : integer;
begin
  write ('Δώσε έναν αριθμό : ');
  readln (x);
  write ('Δώσε ακόμα έναν : ');
  readln (y);
  writeln ('Ο μεγαλύτερος είναι ο : ',Maximum(x,y));
```

end.

Μεγάλα Προγράμματα με Units

Στα μεγάλα προγράμματα η χρήση των units εκτός από τη δόμηση που μας προσφέρουν (παρόμοιες ρουτίνες βρίσκονται όλες μαζί κάπου), προσφέρουν και ταχύτητα κατά το compilation. Η Turbo Pascal προσφέρει τρεις μεθόδους compilation, οι οποίες κάνουν το ίδιο ακριβώς πράγμα όταν γράφουμε μικρά προγράμματα που αποτελούνται από ένα .PAS αρχείο.

Σε ένα μεγάλο πρόγραμμα που αποτελείται από πολλά .PAS αρχεία, ένα κύριο και τα υπόλοιπα units, η επιλογές Compile, Make και Build του Compile menu λειτουργούν ως εξής:

- Η **Compile** κάνει compilation στο αρχείο που έχουμε μπροστά μας στην οθόνη.
- Η **Make** κάνει compilation στο κύριο αρχείο και σε όλα τα units που το .PAS είναι πιο πρόσφατο από το αντίστοιχο .TPU, που σημαίνει ότι έχουν γίνει αλλαγές.
- Η **Build** κάνει compilation σε όλα ανεξαιρέτως τα αρχεία .PAS που αποτελούν το "τεράστιο" πρόγραμμα μας.

ΚΕΦΑΛΑΙΟ 14

Graphics - Γραφικά

Τα γραφικά ήταν πάντοτε ένα χαρακτηριστικό των υπολογιστών που τύγχανε το θαυμασμό όσων τα έβλεπαν, αλλά και την περηφάνεια όσων τα κατασκεύαζαν. Στην Turbo Pascal έχουμε τη δυνατότητα να δημιουργήσουμε γραφικά σε μία μεγάλη ομάδα καρτών γραφικών και με μία πολύ μεγάλη γκάμα από procedures & functions. Επειδή όμως βρισκόμαστε σε μία εποχή που η εξέλιξη, ειδικά στο θέμα των γραφικών είναι μεγάλη, δεν θα σταθώ τόσο πολύ στις επί μέρους ρουτίνες των γραφικών, αλλά σε μία απλή εισαγωγή των γραφικών της Pascal καθώς και σε μία "μαθηματική" προσέγγιση γενικά των γραφικών των υπολογιστών. Αυτό με καλύπτει στην περίπτωση που κάτι αλλάξει στο μέλλον και οι εντολές δεν παραμείνουν οι ίδιες.

Απαραίτητες Εντολές

Οι κάρτες οθόνης που έχουν οι σημερινοί υπολογιστές έχουν τη δυνατότητα να δείχνουν δύο ειδών "εικόνες", αυτές που περιέχουν κείμενο και αυτές που περιέχουν γραφικά. Αυτές οι δύο καταστάσεις ονομάζονται graphic modes. Για να μπορέσουμε να ζωγραφίσουμε με γραφικά στην οθόνη μας, πρέπει να φέρουμε τον υπολογιστή από το text mode που βρίσκεται φυσιολογικά, σε graphics mode. Αφού δε τελειώσουμε με τα γραφικά πρέπει να επαναφέρουμε τον υπολογιστή σε text mode.

Για να "μπούμε" στην κατάσταση γραφικών χρησιμοποιούμε την procedure InitGraph σε συνδυασμό με την DetectGraph. Για να επανέλθουμε χρησιμοποιούμε την CloseGraph. Όλες οι ρουτίνες των γραφικών ανήκουν σε ένα unit, το unit Graph (πρέπει να έχετε το GRAPH.TPU). Επίσης για να μπορέσει η Pascal να φέρει σε κατάσταση γραφικών τον υπολογιστή πρέπει να υπάρχουν τα αρχεία *.BGI στο τρέχον directory (ή όπου αλλού θέλουμε, αλλά καλύτερα στο τρέχον). Όλα αυτά φαίνονται καλύτερα στο παρακάτω παράδειγμα :

```
uses
    Graph, Crt ;
var
    GrDriver, GrMode : integer;
```

Χρησιμοποιούμε και το unit Graph.

Η GrDriver είναι ουσιαστικά ο τύπος της κάρτας γραφικών που έχουμε, ενώ το GrMode είναι η κατάσταση γραφικών στην οποία θα δουλέψουμε (υπάρχουν πολλές σε κάθε κάρτα).

begin

```
DetectGraph (GrDriver,GrMode);
```

```
InitGraph (GrDriver,GrMode, ' ');
```

Στο σημείο αυτό μπορούμε να χρησιμοποιήσουμε όλες τις ρουτίνες γραφικών που θα μάθουμε.

```
CloseGraph;  
end.
```

Βάζουμε τον υπολογιστή να ελέγξει τι κάρτα έχουμε και σε ποιά είναι η μεγαλύτερη διαθέσιμη ανάλυση γραφικών.
"Μπαίνουμε" στην κατάσταση γραφικών. Το κατάλληλο αρχείο BGI θα το φορτώσει από το τρέχον directory ("").

Επανερχόμαστε στην φυσιολογική κατάσταση κειμένου.

Βασικές Έννοιες

Στα γραφικά η οθόνη μας είναι χωρισμένη σε **pixels** (η μικρότερη κουκίδα που μπορεί να ανάψει στην οθόνη). Τα pixels είναι αριθμημένα στον **άξονα των x (οριζόντια)** από δεξιά προς τα αριστερά) και στον **άξονα των y (κάθετα)** από πάνω προς τα κάτω). Η αρίθμηση ξεκινά πάντα από το $(x,y) = (0,0)$ και φτάνει μέχρι την ανάλυση που δουλεύουμε κάθε φορά. Αν π.χ. δουλεύουμε σε ανάλυση 640x480 τότε η αρίθμηση είναι $(0..639 \times 0..479)$.

Όταν δουλεύουμε με τα γραφικά, υπάρχει στην οθόνη, ένας υποθετικός και αόρατος cursor που ονομάζεται **graphics cursor** ή **current pointer (CP)**. Αυτός με την έναρξη της λειτουργίας των γραφικών βρίσκεται στο $(0,0)$ και κατά τη διάρκεια του προγράμματος μας μετακινείται συνέχεια στην τελευταία θέση που κάναμε μία λειτουργία των γραφικών.

Είναι καλό επίσης να γνωρίζεται τις παρακάτω έννοιες. Κάθε στιγμή έχουν κάποια τιμή η οποία αλλάζει με συγκεκριμένες διαδικασίες :

- **Color.** Το χρώμα που χρησιμοποιείται για όλα τα σχήματα (περίγραμμα).
- **Fill Style.** Το στυλ που γεμίζονται όλα τα κλειστά σχήματα (κύκλοι, ελλείψεις, παραλληλόγραμμα κ.λ.π.).
- **Text Style.** Το στυλ των γραμμάτων

Οι Σημαντικότερες Εντολές Γραφικών

Παρακάτω είναι οι **απολύτως σημαντικότερες εντολές**, οι οποίες χρησιμεύουν για τα περισσότερα προγράμματα.

- **procedure PutPixel (x,y : integer; c : color);**

"Ανάβει" ένα pixel στη θέση (x,y) της οθόνης με το χρώμα c.

- **function GetPixel (x,y : integer) : word;**

Επιστρέφει τον αριθμό του χρώματος του pixel στη θέση (x,y).

- **function GetMaxColor : word;**

Επιστρέφει τον αριθμό του μεγαλύτερου χρώματος που μπορούμε να δώσουμε στην SetColor. Π.χ. για το mode 640x480x16 της VGA επιστρέφει 15.

- **procedure SetColor (c : word);**

Θέτει το τρέχον χρώμα των γραφικών. Χρησιμοποιείται για να χρωματίζει γραμμές, κύκλους, παραλληλόγραμμα και γενικά όσα σχήματα έχουν περίγραμμα.

- **procedure Line (x1,y1,x2,y2 : integer);**

Ζωγραφίζει μία γραμμή με τον τρέχον χρώμα από τη θέση (x1,y1) στη θέση (x2,y2).

- **procedure MoveTo (x,y : integer);**

Μετακινεί τον graphics cursor στη θέση (x,y).

- **procedure LineTo (x,y : integer);**

Ζωγραφίζει μία γραμμή από την τρέχουσα θέση του graphics cursor στη θέση (x,y).

- **procedure Rectangle (x1,y1,x2,y2 : integer);**

Ζωγραφίζει το περίγραμμα ενός ορθογωνίου παραλληλογράμμου από τη θέση (x1,y1) στη θέση (x2,y2).

- **procedure Circle (x,y,r : integer);**

Ζωγραφίζει έναν κύκλο με κέντρο (x,y) και ακτίνα r pixels.

- **function GetMaxX : integer;**

Επιστρέφει τη δεξιότερη δυνατή θέση που μπορούμε να ζωγραφίζουμε στην οθόνη μας. Π.χ. για το mode 640x480 επιστρέφει 639.

- **function GetMaxY : integer;**

Δουλεύει όπως και η GetMaxX μόνο που είναι για τον άξονα των y. Στο mode 640x480 θα επιστρέψει 479.

- **procedure ClearDevice;**

Καθαρίζει την οθόνη των γραφικών και επιστρέφει τον graphics cursor στη θέση (0,0).

Παραδείγματα με Graphics

Παρακάτω υπάρχουν κάποια παραδείγματα procedures με graphics. Παραλείπω τα DetectGraph, InitGraph γιατί η χρήση τους είναι ίδια με αυτήν που αναφέρθηκε στις αρχές αυτού του κεφαλαίου. Ετσι, τα παραδείγματα ζητούν να γραφτούν κάποιες procedures οι οποίες δουλεύουν με την προϋπόθεση ότι έχουμε "μπεί" σε κατάσταση γραφικών.

- **Τυχαία Pixels**

Να γίνει μία procedure η οποία τυπώνει τυχαία pixels στην οθόνη με τυχαίο χρώμα και σταματάει όταν πατηθεί κάποιο πλήκτρο.

```
procedure RandomPixels ( Color : byte );
var
  X,Y : integer;
begin
  repeat
    X := Random(GetMaxX+1);
    Y := Random(GetMaxY+1);
    PutPixel (X,Y,Color);
  until KeyPressed;
end;
```

Το GetMaxX δίνει το όριο της οθόνης (639 για την κατάσταση 640x480). Ετσι παίρνουμε Random(GetMaxX+1) για να πάρουμε έναν τυχαίο από 0 έως GetMaxX.
Το ίδιο και για την GetMaxY.

Στο παραπάνω πρόγραμμα αντικαταστήστε την PutPixel με LineTo, Line, Rectangle, Circle κ.λ.π. για να δείτε αντίστοιχα τυχαία "γεμίσματα" της οθόνης.

- **Animation Γραμμής**

Να γίνει μία ρουτίνα που να κινεί μία γραμμή από το κέντρο της οθόνης κυκλικά (όπως η κίνηση του Radar).

Για να πετύχουμε animation πρέπει να προσπαθούμε να έχουμε το "κινητό" ορατό στην οθόνη, για περισσότερο χρόνο απ' όσο είναι αόρατο. Ετσι αφού εμφανίσουμε στην οθόνη αυτό που θέλουμε, καλό είναι να κάνουμε κάποιο delay ώστε να μείνει έστω και κάποια χιλιοστά του δευτερολέπτου περισσότερο ορατό. Πρέπει πάντοτε να ελέγχουμε το χρόνο που απαιτούν οι εντολές που υπάρχουν

μεταξύ της εμφάνισης και της εξαφάνισης του αντικειμένου που μετακινούμε. Σε περιπτώσεις που πρέπει να κάνουμε πολλούς υπολογισμούς (πολ/σμούς - διαιρέσεις κ.λ.π.) καλό είναι να κρατάμε τη θέση του αντικειμένου μας σε κάποιες μεταβλητές, να κάνουμε τους υπολογισμούς της μετακίνησης όσο αυτό είναι ορατό και μετά να σβήνουμε το κινητό μας, από τη θέση στην οποία ήταν. Το παρακάτω παράδειγμα δεν απαιτεί ιδιαίτερους υπολογισμούς και είναι γενικά απλούστατο.

```
procedure AnimateLine;  
var  
    u          : real;  
    r,cx,cy    : integer;  
begin  
    u := 0;  
    cx := GetMaxX div 2;  
    cy := GetMaxY div 2;  
    repeat  
        SetColor (GetMaxColor);  
        MoveTo (cx,cy);  
        LineTo (cx+Round(r*Cos(u)),  
                cy-Round(r*Sin(u)));  
        Delay (10);  
        SetColor (0);  
        LineTo (cx,cy);  
        u := u + 0.01745;  
    until KeyPressed;  
end;
```

Το u είναι η γωνία σε ακτίνια (2π ακτίνια = 360 μοίρες) και r η ακτίνα του κύκλου μέσα στον οποίο κινείται η γραμμή.

Θέτω $u = 0$, (cx,cy) = το κέντρο της οθόνης.

Με άσπρο χρώμα, μετακινούμαι στο κέντρο της οθόνης, και ζωγραφίζω τη γραμμή μέχρι την περιφέρεια του κύκλου, περιμένω 10milliseconds και σβήνω τη γραμμή μέχρι το κέντρο της οθόνης.

Αυξάνω το u κατά 0.01745 που ισοδυναμεί με μία μοίρα. Με μικρότερη αύξηση θα μπορείτε να κινήσετε τη γραμμή πιο αργά.

ΜΕΡΟΣ II

Object Oriented Programming

Κ Ε Φ Α Λ Α Ι Ο 0

Εισαγωγή στον OOP

Ο Object-Oriented Προγραμματισμός (προσανατολισμένος στο αντικείμενο), είναι μία μέθοδος προγραμματισμού, φυσική προέκταση των προηγούμενων μεθόδων των γλωσσών προγραμματισμού. Είναι περισσότερο δομημένος από κάθε προηγούμενη προσπάθεια για δομημένο προγραμματισμό και περισσότερο τμηματικός (modular) και αφηρημένος (abstract) από κάθε προηγούμενη προσπάθεια για απόκρυψη δεδομένων, κώδικα και λεπτομερειών.

Ο "προκάτοχος" του OOP, ο procedural προγραμματισμός, ήταν μία πολύ καλή προσπάθεια για να "σπάσουμε" τον κώδικα του προγράμματός μας σε ανεξάρτητα κομμάτια, τα οποία το ένα "καλούσε" το άλλο. Έτσι είχαμε μία δόμηση η οποία μας βοηθούσε να προλαβαίνουμε διάφορα λάθη, να δουλεύουμε επί πολύ ώρα σε μία procedure διορθώνοντας και βελτιώνοντας την, χωρίς να μας απασχολεί όλο το υπόλοιπο πρόγραμμα. Επίσης έχοντας φτιάξει μία καλή και χρήσιμη ρουτίνα, μπορούσαμε να τη χρησιμοποιήσουμε σαν black box (μαύρο κουτί), δηλαδή χωρίς να μας ενδιαφέρει το πώς ακριβώς έχει γραφτεί. Υπήρχε στο πρόγραμμά μας, αλλά δεν "μπλέκονταν" στα πόδια μας. Οι local μεταβλητές και τα units ήταν επίσης πολύ καλές ιδέες για "απόκρυψη" δεδομένων. Στον προγραμματισμό, απόκρυψη δε σημαίνει ότι είμαστε μυστικοπαθείς και θέλουμε να κρύψουμε τα δεδομένα μας επίτηδες από κάποιον άλλον, αλλά ότι κρύβουμε κάποια κομμάτια του προγράμματος μας από κάποια άλλα, έτσι ώστε να μην υπάρχει κίνδυνος κάποιες ρουτίνες να "χαλάσουν" κάποια δεδομένα που δεν τις αφορούν.

Παρόλες τις προσπάθειες για την βελτίωση του procedural προγραμματισμού, υπήρχε ακόμα ο κίνδυνος κάποια procedure π.χ. να "χαλάσει" την τιμή μιας μεταβλητής, η οποία ήταν global και δεν είχε καμία σχέση με τη συγκεκριμένη ρουτίνα.

Από τη μέχρι τώρα εμπειρία σας θα πρέπει να γνωρίζετε ότι σε ένα πρόγραμμα υπάρχουν κάποιες ρουτίνες που είναι γραμμένες για να χειρίζονται κάποια συγκεκριμένα δεδομένα, κάποιες άλλες για άλλα δεδομένα κ.λ.π. Ο OOP έρχεται ακριβώς σ' αυτό το σημείο να "δέσει" ρουτίνες και δεδομένα σε μία συμπαγή ενότητα, που λέγεται object.

Χαρακτηριστικά του OOP

Τρία είναι τα κύρια χαρακτηριστικά του Object -Oriented Programming :

- ♦ **Encapsulation:** Ο συνδυασμός ενός record με procedures και functions (στο εξής με μία λέξη methods - μέθοδοι) οι οποίες το χειρίζονται, για να δημιουργηθεί ένας νέος τύπος δεδομένων, το object.
- ♦ **Inheritance:** Ο ορισμός ενός object και η χρησιμοποίησή του για να δημιουργηθεί μία ιεραρχία από objects - απογόνους, όπου κάθε απόγονος κληρονομεί όλα τα χαρακτηριστικά των προγόνων του.
- ♦ **Polymorphism:** Μία μέθοδος η οποία μοιράζεται από όλα τα objects μιας ιεραρχίας (κληρονομικής σειράς), όπου κάθε object πραγματοποιεί τη μέθοδο με δικό του τρόπο.

Οι επεκτάσεις της Turbo Pascal με objects, μας δίνουν περισσότερη τμηματοποίηση και "κομμάτιασμα" των προγραμμάτων, μεγαλύτερη απόκρυψη δεδομένων και κώδικα και τη δυνατότητα να μοιραζόμαστε objects με άλλους προγραμματιστές μέσω των units δίνοντας επεκτασιμότητα σε αυτά. Με τον OOP είναι γεγονός ότι μπορούμε να φτιάξουμε ακόμα μεγαλύτερα προγράμματα, με ακόμα μεγαλύτερη ευκολία χειρισμού τους από την πλευρά του προγραμματιστή βέβαια.

Η Πρόκληση και η Δυσκολία

Η δυσκολία του OOP είναι ότι για να μάθουμε και να μπούμε στη φιλοσοφία του, πρέπει να αφήσουμε έξω από το μυαλό μας τις συνήθειες και τους τρόπους σκέψης του προγραμματισμού που ήταν standard για πολλά χρόνια. Από τη στιγμή που θα το κάνουμε, ο OOP είναι μία απλή μέθοδος, ένα πραγματικά εξαιρετικό εργαλείο που λύνει πάρα πολλά προβλήματα που συναντούμε στα παραδοσιακά προγράμματα.

Είναι γεγονός, ότι ακόμα κι αν διαβάσετε και κατανοήσετε πλήρως τον OOP, θέλει λίγο καιρό μέχρι να συνηθίσετε στην ιδέα. Κάθεστε μπροστά στην οθόνη του υπολογιστή σας και σκεπτόμενοι τη λύση ενός προβλήματος, δεν μπορείτε να "χωρέσετε" πουθενά κάποιο object. Είναι απόλυτα φυσιολογικό. Τη στιγμή που προσπαθώντας να λύσετε ένα πρόβλημα, θα σας έλθει η μορφή ενός object στο μυαλό, τότε θα έχετε μπει στο νόημα του OOP και τότε πραγματικά θα ανοίξουν διάπλατα οι πόρτες του νέου αυτού στυλ προγραμματισμού.

Και σε περίπτωση που διαβάζετε αυτές τις γραμμές και δεν γνωρίζετε τίποτα για τον OOP, δεν πειράζει καθόλου. Πολλά λόγια, πολλή σύγχυση και πολλοί άνθρωποι που μιλάνε για κάτι που **δεν** καταλαβαίνουν, έχουν μπερδέψει τον κόσμο τα τελευταία χρόνια. Όλοι θέλουν να μάθουν, όλοι έχουν ακούσει κάτι και νομίζουν ότι έχουν μπει και στο νόημα. Προσπαθήστε να ξεχάσετε τους ανθρώπους που σας έχουν μιλήσει για τον OOP. Ο καλύτερος τρόπος (για την ακρίβεια ο **μόνος** τρόπος) να μάθετε οτιδήποτε χρήσιμο για τον OOP είναι αυτό που θα κάνετε: καθήστε κάτω και δοκιμάστε μόνοι σας. Τη στιγμή που το πρώτο object γεννηθεί εξ' ολοκλήρου μέσα στο μυαλό σας, θα έχετε πετύχει.

Αντικείμενα

Όπου και να κοιτάξετε γύρω σας υπάρχουν αντικείμενα. Δείτε π.χ. το στυλό που έχετε δίπλα σας. Έχει κάποια χαρακτηριστικά. Ας τα αναλύσουμε. Έχει κάποιο χρώμα. Έχει κάποια ποσότητα μελάνης. Έχει ένα ανταλλακτικό. Τι μπορούμε να κάνουμε με το στυλό; Μπορούμε να γράψουμε κάτι. Μπορούμε να αλλάξουμε ανταλλακτικό (= να γεμίσουμε ξανά με μελάνη). Δείτε και το άλλα στυλό (το λίγο πιο επίσημο) που έχετε στο γραφείο σας. Είναι σαν το πρώτο, έχει όμως ένα κουμπάκι για να μπαινοβγαίνει η μύτη του! Ένα καινούργιο λοιπόν χαρακτηριστικό, ανοιχτό/κλειστό και μία κίνηση στο κουμπάκι του, που αλλάζει αυτό το χαρακτηριστικό.

Ετσι και τα αντικείμενα του προγραμματισμού έχουν χαρακτηριστικά (data - πεδία) και πράξεις - κινήσεις - εργασίες πάνω σ' αυτά τα χαρακτηριστικά (methods - procedures & functions) που τα χειρίζονται και τα αλλάζουν. Επίσης, όπως στο παράδειγμα με τα δύο στυλό, έτσι και στα objects το ένα μπορεί να είναι θεωρητικά "εξέλιξη" του άλλου, με πιο πολλά (συνήθως καλύτερα) χαρακτηριστικά και νέες μεθόδους.

Ακόμα, παρατηρείστε στο παράδειγμα ότι το πρώτο στυλό είναι πιο απλό, πιο γενικό. Ετσι και στις κληρονομικές σειρές των objects, όσο προχωρούμε προς τα πάνω, προς τον κύριο πρόγονο δηλαδή, έχουμε έννοιες γενικές με γενικά χαρακτηριστικά. Όσο προχωρούμε προς τα κάτω, οι έννοιες εξειδικεύονται, τα χαρακτηριστικά πληθαίνουν και δίνουν στα objects ξεχωριστές και πιο πολύπλοκες μορφές.

ΚΕΦΑΛΑΙΟ 1

Encapsulation

Ενα object είναι πρακτικά ένα record (ένα σύνολο από δεδομένα διαφορετικών ειδών) εμπλουτισμένο με μεθόδους (procedures και functions) που τα χειρίζονται. Το encapsulation είναι ακριβώς αυτό. Είναι δηλαδή ο τρόπος με τον οποίο γράφουμε τα objects. Ούτε δύσκολο είναι, ούτε απαιτεί καμία ιδιαίτερη πρακτική. Αυτό ίσως που έχει δυσκολία, είναι το πως θα κάνουμε το μυαλό μας να "φτιάξει" ένα object. Με λίγη εξάσκηση όμως και με καλή κατανόηση των παραδειγμάτων, όσο περισσότερα objects μαθαίνουμε τόσο πιο εύκολα θα κατανοήσουμε τη δημιουργία νέων.

Δηλώσεις Object

Τα objects είναι τύποι δεδομένων. Ο καλύτερος τρόπος να μάθουμε γι' αυτά είναι να δούμε ένα παράδειγμα, ενός αντικειμένου που χρησιμεύει για να προσδιορίσει κάποια θέση στην οθόνη με συντεταγμένες x και y :

```
type
  Location = object
    X,Y : integer;

    procedure Init ( NewX,NewY : integer );
    function GetX : integer;
    function GetY : integer;
  end;
```

Είναι λοιπόν ακριβώς παρόμοιο με ένα record μόνο που γράφουμε και τις επικεφαλίδες κάποιων μεθόδων. Οι μέθοδοι αυτές πρέπει να βρίσκονται κάπου παρακάτω γραμμένες ως εξής :

```
procedure Location.Init ( NewX,NewY : integer );
begin
  X := NewX;
  Y := NewY;
end;
```

Στην παραπάνω μέθοδο παρατηρούμε τα εξής :

Το όνομα της μεθόδου περιλαμβάνει και το όνομα του αντικειμένου (Location). Τα X και Y δεν ορίζονται πουθενά μέσα στη μέθοδο Init. Είναι τα πεδία X και Y του object. Μέσα στις μεθόδους λοιπόν ενός object τα πεδία του είναι άμεσα προσβάσιμα. Είναι σα να γράφουμε μέσα σε μία αόρατη with. Επίσης είναι δυνατόν (όπως και στα units) όταν γράφουμε το

"σώμα" της μεθόδου να **μην** γράψουμε την πλήρη επικεφαλίδα αλλά απλά το όνομα του object, τελεία (.) και το όνομα της μεθόδου. Στην πράξη όμως βολεύει να ξαναγράψουμε την επικεφαλίδα, για να βλέπουμε από κοντά τις παραμέτρους που παίρνει η κάθε μέθοδος.

Η μέθοδος Init που φτιάξαμε, όπως είναι φανερό, "δέχεται" από τον έξω κόσμο δύο ακεραίους και τους τοποθετεί στα πεδία X και Y του αντικειμένου.

Για να χρησιμοποιήσουμε τώρα το παραπάνω object πρέπει να φτιάξουμε μία μεταβλητή αυτού του τύπου. Να κάνουμε ένα **instance του object** όπως λέμε. Όταν λέμε "έχω δύο instances του τάδε object" εννοούμε ότι έχουμε δύο μεταβλητές αυτού του τύπου δεδομένων. Έτσι λοιπόν χρησιμοποιούμε το object Location γράφοντας :

```
var
  OneLocation : Location;
begin
  OneLocation.Init (100,50);
  writeln (OneLocation.GetX);
end.
```

Γράφοντας OneLocation.Init καλούμε τη μέθοδο Init της μεταβλητής OneLocation δηλαδή του τύπου Location. Αυτή θα βάλει τους αριθμούς 100 και 50 στα πεδία X και Y του αντικειμένου μας (OneLocation) αντίστοιχα. Αν είχαμε records θα γράφαμε OneLocation.X := 100 και OneLocation.Y := 50. Αυτό μπορούμε να το κάνουμε **και** με τα objects, αλλά γενικά δεν είναι καλή προγραμματιστική τεχνική.

Η ρουτίνα OneLocation.GetX επιστρέφει την τιμή του πεδίου X, δηλαδή πρέπει να μας τυπώσει 100. Θα μπορούσαμε και εδώ να γράφαμε writeln (OneLocation.X) όπως ακριβώς αν το X ήταν πεδίο ενός record. Όμως και εδώ δεν είναι καλή συνήθεια. Η ρουτίνα GetX θα γραφόταν κάπως έτσι :

```
function Location.GetX : integer;
begin
  GetX := X;
end;
```

To Νόημα του Encapsulation

Γενικά, τα objects πρέπει να τα βλέπουμε σαν συμπαγείς σφαίρες, όπου στο κέντρο τους υπάρχει ένας πυρήνας. Στον πυρήνα ανήκουν τα πεδία = τα δεδομένα του objects. Γύρω από τον πυρήνα, κυκλοφορούν διάφορες μέθοδοι. Η μέθοδοι είναι οι μόνες ρουτίνες που μπορούν να αλλάξουν τα δεδομένα ενός object! Δεν είναι καθόλου σωστό (αν και

επιτρέπεται) να πάμε μόνοι μας και να χειριστούμε ή να αλλάξουμε τα δεδομένα αυτά, όπως κάναμε με τα records. Οτι θέλουμε να αλλάξουμε, το ζητάμε από μία μέθοδο. Οτι θέλουμε να πάρουμε το ζητάμε επίσης από μία μέθοδο.

Οι μέθοδοι μπορούν να "συνεργάζονται" μεταξύ τους. Μία μέθοδος μπορεί να καλεί μία άλλη. Επίσης είναι δυνατόν, κάποια από τα στοιχεία του object (πεδία - μεθόδους) να τα κρύψουμε από τον χρήστη κάνοντάς τα private (προσωπικά). Τότε αυτά, μπορούν να τα χειριστούν μόνοι οι μέθοδοι του ίδιου του object και κανείς άλλος.

Τέλος, είναι μία καλή τεχνική να γράφουμε τα objects μας μέσα σε units. Στο interface κομμάτι ενός unit γράφουμε τη δήλωση του object και στο implementation, υλοποιούμε όλες τις μεθόδους του. Επίσης αν το object κληρονομείται (βλ. επόμενο κεφάλαιο), είναι καλό στο ίδιο unit να έχουμε και όλους τους απογόνους του. Ετσι χρησιμοποιώντας αυτό το unit θα έχουμε πρόσβαση σε όλα τα objects μίας κληρονομικής σειράς.

Ενα Συγκριτικό Παράδειγμα

- **Τυχαίες γραμμές στην οθόνη**

Εστω ότι θέλουμε να ζωγραφίσουμε τυχαίες γραμμές στην οθόνη μέχρι να πατηθεί ένα πλήκτρο. Παραλείποντας τις εντολές που απαιτούνται για την έναρξη και λήξη των γραφικών, δείτε παρακάτω σε δύο στήλες δύο προγράμματα που λύνουν την άσκηση. Το πρώτο είναι σε Παραδοσιακή Pascal, το δεύτερο σε Object Pascal.

```

program TraditionalLines;

var
    x1,y1,x2,y2 : integer;

begin
    repeat
        x1 := Random(640);
        y1 := Random(480);
        x2 := Random(640);
        y2 := Random(480);
        Line (x1,y1,x2,y2);
    until KeyPressed;
end.

program ObjectLines; { Wow! }

type
    OneLine = object
        x1,y1,x2,y2 : integer;

        procedure SetRandom;
        procedure Draw;
    end;

    procedure OneLine.SetRandom;
begin
    x1 := Random(640);
    y1 := Random(480);
    x2 := Random(640);
    y2 := Random(480);
end;

    procedure OneLine.Draw;
begin
    Line (x1,y1,x2,y2);
end;

var
    MyLine : OneLine;

begin
    repeat
        MyLine.SetRandom;
        MyLine.Draw;
    until KeyPressed;
end.

```

Όπως φαίνεται πολύ καθαρά, το δεύτερο είναι πιο μεγάλο. Πρώτο λοιπόν συμπέρασμα ότι όταν κάνουμε Object Oriented Προγραμματισμό, γράφουμε περισσότερα. **ΛΑΘΟΣ!** Στο συγκεκριμένο παράδειγμα, φαίνεται ότι γράφουμε περισσότερα και πραγματικά έτσι είναι, μέχρι όμως να αποφασίσουμε να ζωγραφίζουμε δύο γραμμές ταυτόχρονα. Γιατί τότε στο πρώτο πρόγραμμα, θα προσθέταμε : τέσσερις νέες μεταβλητές (με το ίδιο ακαταλαβίστικα ονόματα), τέσσερις γραμμές για να δίνουν τιμές σ' αυτές και μία γραμμή για να ζωγραφίζει. Στο πρόγραμμα με τα objects όμως, θα προσθέταμε μία μεταβλητή, μία γραμμή για να δώσουμε τιμές και μία γραμμή για να ζωγραφίσουμε.

Γενικά δεν πρέπει να κρίνουμε τα πράγματα, όπως αυτά φαίνονται στα παραδείγματα, γιατί τα παραδείγματα είναι αναγκαστικά μικρά και συνήθως άχρηστα προγραμματάκια. Σε ένα μεγάλο πρόγραμμα ο object oriented προγραμματισμός είναι πολύ πιο σύντομος. Στο επόμενο κεφάλαιο της κληρονομικότητας θα καταλάβουμε ακόμα καλύτερα πόσο πιο σύντομα προγραμματίζουμε χρησιμοποιώντας objects.

Ας δούμε λίγο συγκριτικά το παραπάνω παράδειγμα :

- **Απόκρυψη.** Φανταστείτε το object να ήταν γραμμένο σε κάποιο unit. Θα ήταν το κυρίως πρόγραμμά μας πιο σύντομο;
- **Μεταβλητές.** Προσέξτε ότι στο πρώτο πρόγραμμα χρησιμοποιούμε 4 μεταβλητές ενώ στον OOP μόνο μία. Και είναι γνωστό ότι με όσο περισσότερες μεταβλητές ή procedures ασχολούμαστε τόσο το χειρότερο.
- **Αναγνωσιμότητα.** Προσέξτε πόσο πιο κατανοητό είναι το δεύτερο πρόγραμμα. Το διαβάζουμε σαν να ήταν απλά αγγλικά. Εδώ φαίνεται πόσο πιο High - Level γίνεται ο προγραμματισμός με objects.
- **Object Oriented.** Προσέξτε πως είναι το πρόγραμμα προσανατολισμένο στο αντικείμενο. Οτι γράφουμε το γράφουμε δίνοντας το όνομα του αντικειμένου μπροστά. MyLine.SetRandom. Στη "γραμμή μου" βάλε τυχαίες τιμές. MyLine.Draw. Το MyLine ζωγράφισε το.

Συμπέρασμα

Σ' αυτό το κεφάλαιο γνωρίσαμε το πρώτο μέρος του Object Oriented Programming. Ουσιαστικά είδαμε, πως γράφεται ένα αντικείμενο και πως χρησιμοποιείται. Αυτό το οποίο μπορούμε να συμπεράνουμε αυτή τη στιγμή, είναι ότι στον OOP έχουμε να κάνουμε με κάτι πιο High Level, με κάτι που μοιάζει περισσότερο στη φιλοσοφία μας, με κάτι που αποκρύπτει μεγάλα κομμάτια του προγράμματός μας και τα βάζει σε καλούπια. Ένα object πρέπει να το βλέπετε σαν μία σφαίρα μεταλλική, συμπαγή, κλειστή και δυνατή. Κρύβει μέσα του τόσες δυνάμεις, μπορεί να αναπτυχθεί και είναι σίγουρο ότι στα χέρια του ικανού προγραμματιστή θα αποτελέσει το καλύτερο εργαλείο που έχει εφευρεθεί μέχρι σήμερα.

Ο Object Oriented προγραμματισμός, είναι μία ακριβής ανταπόκριση στις ανάγκες του μοντέρνου προγραμματισμού και στην πολυπλοκότητα των σημερινών προγραμμάτων που έχουν κάνει ακόμα και έμπειρους προγραμματιστές να "σηκώσουν ψηλά τα χέρια τους". Η κληρονομικότητα και το encapsulation είναι τεράστιας σημασίας εργαλεία, για να μπορέσουμε να ελέγξουμε την πολυπλοκότητα αυτή. Για να δανειστώ και μία φράση από τα manuals της Borland Pascal : "Είναι η διαφορά μεταξύ, του να έχεις 10,000 έντομα τακτοποιημένα σε ένα διάγραμμα ταξινόμησης, από το να έχεις 10,000 έντομα να στριφογυρίζουν γύρω από το κεφάλι σου"...

ΚΕΦΑΛΑΙΟ 2

Inheritance - Κληρονομικότητα

Η κληρονομικότητα, είναι ίσως το πιο βασικό χαρακτηριστικό του OOP. Είναι αυτό που δίνει την επεκτασιμότητα στα objects, είναι αυτό που μας δίνει τη δυνατότητα να μην ξαναγράφουμε διπλό και τριπλό κώδικα αλλά να χρησιμοποιούμε όσο μπορούμε περισσότερο κώδικα που υπάρχει. Γενικά η κληρονομικότητα είναι αυτό που θα μας δείξει ότι τελικά στον OOP, γράφουμε λιγότερα. Με πιο σύντομα προγράμματα καταφέρνουμε να κάνουμε περισσότερα πράγματα.

Ενα αντικείμενο (απόγονος - descendant) που κληρονομεί ένα άλλο (πρόγονος - ancestor), κρατάει όλα τα χαρακτηριστικά (πεδία) του προγόνου του, καθώς και τις μεθόδους του. Μπορούμε όμως να του δώσουμε πρόσθετα χαρακτηριστικά, πρόσθετες μεθόδους, ενώ τέλος μπορούμε να ξαναγράψουμε ή να βελτιώσουμε κάποιες μεθόδους που κληρονομήθηκαν.

Δηλώσεις Descendant Objects (Απογόνων)

Για να καταλάβουμε καλύτερα πως μπορούμε να φτιάξουμε ένα object απόγονο, ενός άλλου που ήδη υπάρχει, ας προσπαθήσουμε να φτιάξουμε έναν απόγονο του αντικειμένου Location που χρησιμοποιήθηκε σαν παράδειγμα στο προηγούμενο κεφάλαιο. Ξαναγράφουμε τη δήλωση του Location για να υπάρχει καλύτερη εικόνα :

```
type
  Location = object
    X,Y : integer;

    procedure Init ( NewX,NewY : integer );
    function GetX : integer;
    function GetY : integer;
  end;
  Pixel = object (Location)
    Color : byte;

    procedure Init ( NewX,NewY : integer; NewColor : byte );
    function GetColor : byte;
    procedure Show;
  end;
```

Οπως βλέπουμε το object Pixel είναι ένας απόγονος του Location με μία επιπλέον ιδιότητα. Το χρώμα (Color) που είναι byte. Για να χρησιμοποιηθεί σωστά το Pixel, προστίθενται δύο

μέθοδοι, η GetColor που μας επιστρέφει το χρώμα που έχει το Pixel και η Show που το εμφανίζει στην οθόνη. Το object Pixel έχει τώρα τρία πεδία. Τα X και Y που κληρονόμησε από το Location και το Color. Επίσης έχει μεθόδους την GetX, την GetY, την GetColor την Show και την Init. Αλήθεια, τι γίνεται με την Init; Βλέπουμε ότι το Pixel κληρονομεί την Init από το Location, όμως υπάρχει ξανά η δήλωση της Init στο Pixel και μάλιστα αλλαγμένη.

Η Object Pascal μας δίνει τη δυνατότητα να ξαναγράψουμε μία μέθοδο που έχουμε κληρονομήσει. Αυτό ονομάζεται **method override** και είναι ένα από τα ωφέλη της κληρονομικότητας. Επειδή η Init που κληρονομήσαμε δε μας βόλεψε, έχουμε τη δυνατότητα να φτιάξουμε μία άλλη με το ίδιο όνομα. Αυτό δε δημιουργεί προβλήματα στη γλώσσα, γιατί απλούστατα η μία Init είναι του Location και η άλλη είναι του Pixel. Μ' αυτόν τον τρόπο, όπως καταλαβαίνετε μπορούμε να χρησιμοποιούμε το ίδιο όνομα σε μία κληρονομική σειρά για μεθόδους που κάνουν το ίδιο περίπου πράγμα, κάτι πολύ καλό, αν σκεφτούμε πως θα γινόταν ο προγραμματισμός αν για κάθε αντικείμενο έπρεπε να χρησιμοποιούμε διαφορετικά ονόματα μεθόδων.

Ετσι, λοιπόν συνηθίζουμε να έχουμε την Init για να βάζει αρχικές τιμές στα πεδία ενός αντικειμένου, τη Show για να το δείχνει στην οθόνη, τη Hide για να το "κρύβει" - σβήνει από την οθόνη κ.λ.π.

Κλήση Μεθόδων

Στην Object Pascal (η επέκταση της Pascal με objects) μπορούμε να καλέσουμε από μία μέθοδο ενός object μία άλλη χρησιμοποιώντας απλά το όνομά της. Θυμηθείτε ότι μέσα στις μεθόδους ενός object αναφερόμαστε στα πεδία του, με το όνομά τους σαν να ήταν τοπικές (local) μεταβλητές. Το ίδιο συμβαίνει και με τις μεθόδους. Για παράδειγμα ας φανταστούμε ότι το Pixel είχε και μία μέθοδο Hide που το έσβηνε απ' την οθόνη και μία μέθοδο Move (vx,vy) που το μετακινούσε στη θέση vx,vy της οθόνης. Πως θα γράφαμε τη Move;

```
procedure Pixel.Move ( vx,vy : integer );
begin
  Hide;
  X := vx;
  Y := vy;
  Show;
end;
```

Όπως βλέπουμε η Move καλεί τη Hide για να "κρύψει" το αντικείμενο, τοποθετεί νέες τιμές στα X και Y, και τέλος καλεί τη Show για να εμφανίσει και πάλι το αντικείμενο στην καινούργια θέση της οθόνης.

Inherited (Κληρονομημένες) Μέθοδοι

Πολύ συχνά, γράφοντας μία αλλαγμένη μέθοδο χρειαζόμαστε τη λειτουργικότητα της μεθόδου που κληρονομήσαμε. Στην Turbo Pascal v7.0 και μετά μπορούμε να καλέσουμε από μία μέθοδο ενός object, μία μέθοδο του προγόνου του, χρησιμοποιώντας τη λέξη inherited. Συνήθως αυτό συμβαίνει όταν κάνουμε method override.

Για παράδειγμα ας δούμε παρακάτω δύο τρόπους για να γράψουμε την Init του αντικειμένου Pixel.

```
{ Η παρακάτω δεν είναι εντελώς σωστή μέθοδος. Μπορεί να βελτιωθεί }  
procedure Pixel.Init ( NewX,NewY : integer; NewColor : byte );  
begin  
    X := NewX;  
    Y := NewY;  
    Color := NewColor;  
end;
```

Στην παραπάνω μέθοδο, απλά και καθαρά, βάζουμε τις τρεις νέες τιμές στα αντίστοιχα πεδία του αντικειμένου Pixel, τα X και Y που κληρονόμησε και το Color που είναι καθαρά δικό του. Δεν είναι όμως κουραστικό να κάνουμε την ίδια δουλειά για τα πεδία X και Y που κάνει η Init του Location, δηλαδή η Init που έχουμε κληρονομήσει; Ας σκεφτούμε επίσης ότι το Location είχε δύο μόλις πεδία και γενικά η Init έκανε μικρή δουλειά. Τι γίνεται όμως στις περιπτώσεις που η Init που έχουμε κληρονομήσει είναι τεράστια και εμείς θέλουμε να προσθέσουμε ουσιαστικά μία - δύο γραμμές;

```
procedure Pixel.Init ( NewX,NewY : integer; NewColor : byte );  
begin  
    inherited Init (NewX,NewY);  
    Color := NewColor;  
end;
```

Στο τελευταίο παράδειγμα βλέπουμε ότι η Init του Pixel καλεί την κληρονομημένη Init για να δώσει τιμές στα πεδία X και Y και βάζει μόνη της τη νέα τιμή στο πεδίο Color. Περίττο βέβαια να πούμε ότι αυτό είναι το σωστό παράδειγμα και ο σωστός Object Oriented προγραμματισμός.

Στις προηγούμενες εκδόσεις της Turbo Pascal (πριν την 7.0) όπου δεν υπήρχε η λέξη `inherited` μπορούσαμε να καλέσουμε την κληρονομημένη μέθοδο γράφοντας το όνομα του αντικειμένου - προγόνου. Έτσι αντί `inherited Init (...)` θα γράφαμε `Location.Init (...)`

Units με Αντικείμενα

Είναι μία καλή ιδέα να ορίζουμε `objects` μέσα σε `Units`, γράφοντας τη δήλωση `type` του αντικειμένου στο `interface` μέρος του `Unit` και τις μεθόδους που αναφέρονται στο αντικείμενο στο `implementation` μέρος.

Μέσα στα `Units` μπορούμε να έχουμε και άλλα κρυφά αντικείμενα, ή κρυφές βοηθητικές ρουτίνες που θα βοηθούν στην υλοποίηση του αντικειμένου που "εξάγουμε". Ακόμα, μπορούμε να βάλουμε και μία ολόκληρη κληρονομική σειρά με 5-6 αντικείμενα σε ένα `Unit` έτσι ώστε όποιος το χρησιμοποιεί να μπορεί να έχει πρόσβαση σε όλα αυτά τα αντικείμενα.

Σε κάποιες περιπτώσεις, θέλουμε να εξάγουμε κάποιο `object` αλλά δε θέλουμε να δώσουμε πρόσβαση σε όλα τα πεδία ή μεθόδους του αντικειμένου. Για παράδειγμα στο `object Pixel` δε θέλουμε ο προγραμματιστής που θα το χρησιμοποιήσει (ακόμα κι εμείς) να έχει απ' ευθείας πρόσβαση στα πεδία `X,Y,Color` αλλά μόνο στις μεθόδους που τα χειρίζονται.

Από την έκδοση 7.0 της Turbo Pascal έχουμε στη διάθεση μας τις λέξεις **`public`** (κοινοποιημένο) και **`private`** (ιδιωτικό - μυστικό). Είναι δύο λέξεις που χρησιμοποιούνται μέσα στις δηλώσεις των `objects` και μας δίνουν τη δυνατότητα να ορίσουμε ποιά από τα πεδία ή μέθοδοι, θέλουμε να είναι κρυμμένα ή φανερά. Εξ' ορισμού τα πεδία και οι μέθοδοι ενός αντικειμένου είναι `public`.

Όταν ένα πεδίο ή μέθοδος είναι **`private`** τότε μπορεί να χρησιμοποιηθεί (ή να κληθεί) μόνο μέσα στο `module` (πρόγραμμα ή `unit`) που είναι δηλωμένο το αντικείμενο. Όταν είναι `public` μπορεί να χρησιμοποιηθεί από οποιοδήποτε `module` χρησιμοποιεί το `module` που είναι δηλωμένο το αντικείμενο. Έτσι λοιπόν, γίνεται φανερό ότι αν έχω μία κληρονομική σειρά κάποιων αντικειμένων σε ένα `unit` τότε τα `private` πεδία και μέθοδοι τους, μπορούν να χρησιμοποιηθούν οπουδήποτε μέσα σ' αυτό το `unit` αλλά όχι έξω απ' αυτό. Ο ακριβής τρόπος χρήσης των λέξεων `public` και `private` περιγράφεται στο παρακάτω παράδειγμα.

Παράδειγμα με Inherited Objects

Στο παράδειγμα που ακολουθεί εξηγούνται πλήρως όσα αναφέρθηκαν στην προηγούμενη παράγραφο.

- **Unit με τα αντικείμενα Location και Pixel**

Εδώ θα γράψουμε ένα πλήρες Unit που θα περιέχει τα αντικείμενα Location και Pixel σε μία ανεπτυγμένη μορφή με όλα όσα μάθαμε μέχρι στιγμής. Προσέξτε τις δηλώσεις των δύο αντικειμένων στο interface section του Unit και τις μεθόδους τους στο implementation section. Επίσης προσέξτε τους ορισμούς public και private.

```
unit MyObjects;

interface

type
    Location = object
        private
            X,Y : integer;
        public
            procedure Init ( NewX,NewY : integer );
            function GetX : integer;
            function GetY : integer;
        end;
    Pixel = object (Location)
        private
            Color : byte;
        public
            procedure Init ( NewX,NewY : integer; NewColor : byte );
            function GetColor : byte;
            procedure Show;
            procedure Hide;
            procedure Move ( NewX,NewY : integer );
        end;

implementation

{ ---- Μέθοδοι του Location ----}
procedure Location.Init ( NewX,NewY : integer );
begin
    X := NewX;
    Y := NewY;
end;

function Location.GetX : integer;
begin
    GetX := X;
end;
```



```

function Location.GetY : integer;
begin
    GetY := Y;
end;

{ ---- Μέθοδοι του Pixel ---- }

procedure Pixel.Init ( NewX,NewY : integer; NewColor : byte );
begin
    inherited Init (NewX,NewY);
    Color := NewColor;
end;

function Pixel.GetColor : byte;
begin
    GetColor := Color;
end;

procedure Pixel.Show;
begin
    PutPixel (X,Y,Color);
end;

procedure Pixel.Hide;
begin
    PutPixel (X,Y,0);
end;

procedure Pixel.Move ( NewX,NewY : integer );
begin
    Hide;
    X := NewX;
    Y := NewY;
    Show;
end;

end.

```

Polymorphism - Πολυμορφισμός

Ο πολυμορφισμός, λέξη ελληνική (πολλές μορφές) είναι το τρίτο μεγάλο κεφάλαιο του Object Oriented προγραμματισμού. Είναι ίσως και το δυσκολότερο. Μέχρι στιγμής είδαμε το δύο πρώτα χαρακτηριστικά του OOP, το **encapsulation** που δεν είναι τίποτε άλλο από το "δέσιμο" πεδίων και μεθόδων σε μία δομή που ονομάζεται object και την **κληρονομικότητα (inheritance)** που δίνει τη δυνατότητα σε ένα object να είναι απόγονος ενός άλλου, κερδίζοντας όλα τα χαρακτηριστικά του προγόνου του.

Ο πολυμορφισμός έρχεται να λύσει ένα πρόβλημα. Ένα πρόβλημα που συναντάται σε μία κληρονομική σειρά. Το πρόβλημα αυτό θα προσπαθήσουμε να αναλύσουμε στην επόμενη παράγραφο.

Κληρονομώντας Στατικές Μεθόδους

Στο προηγούμενο κεφάλαιο είδαμε ένα αντικείμενο Pixel (που κληρονομούσε το Location) και που είχε σαν χαρακτηριστικά τη θέση του στην οθόνη (x,y), το χρώμα του (Color) και είχε τη μέθοδο Init που έθετε νέες τιμές στα πεδία του, τη Show που το εμφάνιζε στην οθόνη, τη Hide που το έσβηνε από την οθόνη και τη MoveTo που το μετακινούσε σε κάποια άλλη θέση.

Για λόγους συντομίας θα ξαναγράψουμε εδώ το object Pixel λίγο αλλαγμένο, χωρίς public και private μέρη και χωρίς τη function GetColor που δε θα μας χρειαστεί :

```
TPixel = object (Location)
    Color : byte;

    procedure Init ( NewX,NewY : integer; NewColor : byte );
    procedure Show;
    procedure Hide;
    procedure MoveTo ( NewX,NewY : integer );
end;
```

Το object TPixel λοιπόν είναι ένα pixel της οθόνης μας, που μπορούμε να το αναβοσβήνουμε και να το μετακινούμε.

Ας φτιάξουμε τώρα έναν απόγονο του TPixel, το αντικείμενο TCircle το οποίο είναι ένας κύκλος με κέντρο X,Y, χρώμα Color και (το νέο χαρακτηριστικό του) ακτίνα Radius.

```
TCircle= object (TPixel)
    Radius : integer;

    procedure Init ( NewX,NewY,NewR : integer; NewColor : byte);
    procedure Show;
    procedure Hide;
    procedure MoveTo ( NewX,NewY : integer );
end;
```

Στο TCircle ξαναγράφουμε την Init για να θέτουμε και την ακτίνα του Κύκλου. Η TCircle.Show δείχνει τον κύκλο στην οθόνη (και όχι ένα μόνο pixel όπως η Show που κληρονομήσαμε), η TCircle.Hide σβήνει τον κύκλο από την οθόνη και η TCircle.MoveTo μετακινεί τον κύκλο σε κάποιο άλλο μέρος της οθόνης.

Οι δύο μέθοδοι Show λοιπόν διαφέρουν, όπως και οι δύο Hide. Αυτό είναι προφανές. Τι γίνεται όμως με τις δύο MoveTo; Ας δούμε καλύτερα πως θα ήταν γραμμένες αυτές οι δύο μέθοδοι :

```
procedure TPixel.MoveTo ( NewX,NewY : integer );
begin
    Hide;           { κρύβει το Pixel }
    X := NewX;
    Y := NewY;
    Show;          { εμφανίζει το Pixel στη νέα θέση }
end;

procedure TCircle.MoveTo ( NewX,NewY : integer );
begin
    Hide;           { κρύβει τον κύκλο }
    X := NewX;
    Y := NewY;
    Show;          { εμφανίζει τον κύκλο στη νέα θέση }
end;
```

Όπως φαίνεται, τίποτα δεν αλλάζει. Μπορούμε δηλαδή απλά να αντιγράψουμε τη μία ρουτίνα στην άλλη και να αλλάξουμε απλώς το όνομα του αντικειμένου που του ανήκει. Το TPixel να το κάνουμε TCircle.

Αρα γιατί να μην κρατήσουμε τη MoveTo που κληρονομήσαμε χωρίς να την ξαναγράψουμε για τον κύκλο, αφού είναι η ίδια; Η απάντηση είναι απλή. Η MoveTo που κληρονομήσαμε, κρύβει και εμφανίζει Pixels. Αρα μας είναι άχρηστη.

Το πρόβλημα λοιπόν βρίσκεται ακριβώς εδώ. Αν δεν ξαναγράψουμε τη MoveTo για το TCircle δε θα μπορούμε να μετακινούμε τους κύκλους. Η λύση ήταν απλή. Και θα

μπορούσαμε να την αφήναμε έτσι. Όμως, τι θα γινόταν αν είχαμε δέκα διαφορετικά αντικείμενα απογόνους του Pixel, κάθε ένα με διαφορετικές Show και Hide; Τι θα γινόταν αν η MoveTo δεν ήταν τόσο μικρή (4 γραμμές), αλλά μία τεράστια ρουτίνα των 100 γραμμών; Γιατί να πρέπει να έχουμε μέσα στο πρόγραμμά μας αυτές τις ολόιδιες γραμμές, επαναλαμβανόμενες αρκετές φορές;

Οι μέθοδοι που είδαμε μέχρι στιγμής ήταν στατικές (static). Ονομάζονται έτσι, όπως και οι στατικές μεταβλητές (σε αντίθεση με τις δυναμικές - pointers) γιατί απλούστατα, είναι γνωστές από τον Compiler τη στιγμή που γίνεται το Compilation. Όταν ο Compiler μεταφράζει τη ρουτίνα TPixel.MoveTo στο σημείο που καλείται η Hide θα καλέσει την TPixel.Hide και στο σημείο που εμείς γράφουμε Show θα καλέσει την TPixel.Show. Όλα αυτά είναι γνωστά από την κανονική Pascal.

Virtual Methods

Οι μέθοδοι που δεν είναι static είναι virtual (μη πραγματικές - ιδεατές - φανταστικές). Οι μέθοδοι αυτές έρχονται για να δώσουν το μεγαλύτερο μέρος της δύναμης στον OOP, έρχονται για να λύσουν το πρόβλημα που τέθηκε στην προηγούμενη παράγραφο.

Οι Hide και Show στο προηγούμενο παράδειγμα, είναι ρουτίνες οι οποίες μοιράζονται σε όλα τα αντικείμενα της κληρονομικής σειράς (υποθέστε ότι υπάρχουν και άλλα). Σε κάθε αντικείμενο όμως κάνουν διαφορετική δουλειά - ζωγραφίζουν άλλο σχήμα! Έχουν δηλαδή πολλές μορφές. Είναι πολυμορφικές.

Η λύση στο πρόβλημα μας έρχεται αν ορίσουμε ότι η Show και η Hide είναι virtual μέθοδοι. Αν γίνει αυτό τότε μπορούμε να κληρονομήσουμε τη MoveTo από το TPixel στο TCircle χωρίς **καμία απολύτως αλλαγή**. Η TPixel.MoveTo τώρα, δεν θα καλεί την TPixel.Show και την TPixel.Hide αλλά τις Show και Hide του αντικειμένου που κάλεσε τη MoveTo (είτε αυτό είναι TPixel, είτε TCircle, είτε οποιοσδήποτε άλλος απόγονος του TPixel)!

Πραγματικά φανταστικό το αποτέλεσμα, αλλά πως γίνεται αυτό; Απλώς όταν ο Compiler πρέπει να καλέσει μία virtual μέθοδο, ουσιαστικά πάει να καλέσει μία μέθοδο που δεν υπάρχει εκείνη τη στιγμή. Έτσι σε εκείνο το σημείο αφήνει κάτι σαν εκκρεμότητα. Η εκκρεμότητα αυτή θα λυθεί όταν το πρόγραμμα τρέξει, όπου σε εκείνο το σημείο θα κληθεί η κατάλληλη ρουτίνα.

Όταν η MoveTo αρχίζει να δουλεύει, το πρόγραμμα ξέρει ποιό ακριβώς αντικείμενο κάλεσε τη ρουτίνα αυτή. Ξέρει επίσης αν αυτό το αντικείμενο είναι ένα TPixel ή κάποιος απογόνος του TPixel (όπως το TCircle) που την κληρονόμησε. Έτσι, στο σημείο όπου καλείται η Hide, θα κληθεί εκείνη τη στιγμή η αντίστοιχη Hide του αντικειμένου που κάλεσε τη MoveTo.

Οι μόνες αλλαγές που πρέπει να κάνουμε στον κώδικα μας, είναι στις δηλώσεις των objects. Έτσι τα TPixel και TCircle θα ξαναγραφτούν ως εξής :

```
TPixel = object (Location)
    Color : byte;

    constructor Init ( NewX,NewY : integer; NewColor : byte );
    procedure Show; virtual;
    procedure Hide; virtual;
    procedure MoveTo ( NewX,NewY : integer );
end;
TCircle= object (TPixel)
    Radius : integer;

    constructor Init ( NewX,NewY,NewR:integer; NewColor:byte);
    procedure Show; virtual;
    procedure Hide; virtual;
end;
```

Προσέξτε τις λέξεις virtual στο τέλος των δηλώσεων των Show και Hide και στα δύο objects. Επίσης προσέξτε ότι η λέξη procedure αντικαταστάθηκε με τη λέξη constructor. Ένας constructor (κατασκευαστής) είναι μία μέθοδος η οποία είναι απαραίτητη για να δουλέψουν τα αντικείμενα με virtual methods. Γενικά ισχύουν οι παρακάτω κανόνες :

- Από τη στιγμή που κάποια ρουτίνα ενός object είναι virtual, θα παραμείνει virtual σε όλους τους απογόνους του object αυτού.
- Κάθε object που έχει virtual methods, χρειάζεται απαραίτητα έναν constructor.
- Πριν εκτελεστεί οποιαδήποτε virtual method κάποιας μεταβλητής τύπου object, πρέπει να έχει εκτελεστεί ο constructor της.

Προσέξτε ότι στον τελευταίο κανόνα μιλάμε για μεταβλητές. Μην ξεχνάτε ποτέ ότι τα TPixel και TCircle είναι τύποι δεδομένων. Όταν θα θελήσετε να τα χρησιμοποιήσετε σε κάποιο πρόγραμμα θα πρέπει να φτιάξετε μεταβλητές αυτών των τύπων. Έτσι, αν θέλουμε να έχουμε δύο μεταβλητές του τύπου TPixel στο πρόγραμμα μας, και θέλουμε να καλέσουμε τη Show για να τα εμφανίσουμε, πρέπει να καλέσουμε την Init **και για τις δύο** μεταβλητές!

Στο παρακάτω παράδειγμα φαίνεται η χρήση των μεταβλητών (instances) τύπου TPixel και TCircle :

var	Υποτίθεται ότι προηγούνται οι δηλώσεις των TPixel και TCircle με τις virtual methods...
A : TPixel; B : TCircle;	
begin	
A.Init (100,100,15);	Δίνουμε τιμές στο A. Θέση 100,100. Χρώμα 15.
A.Show;	Εμφανίζουμε το pixel A στην οθόνη.
B.Init (200,200,30,14);	Δίνουμε τιμές στον κύκλο B.
B.Show;	Εμφανίζουμε τον κύκλο B στην οθόνη.
A.MoveTo (150,150);	Καλούμε τη MoveTo για το A. Αυτή είναι η TPixel.MoveTo. Επειδή το A είναι TPixel θα κληθούν οι TPixel.Hide και TPixel.Show για να σβήσουν το pixel και να το εμφανίσουν στη νέα του θέση (150,150).
B.MoveTo (10,10);	Καλούμε τη MoveTo για το B. Αν και το B είναι TCircle, καλείται η κληρονομημένη MoveTo, δηλαδή η TPixel.MoveTo, η οποία όμως θα καλέσει τις TCircle.Hide και TCircle.Show για να σβήσουν και να εμφανίσουν τον κύκλο στη νέα του θέση (10,10)
end.	

Constructors

Κάθε αντικείμενο έχει έναν Virtual Method Table (VMT) που είναι ένας πίνακας που περιέχει τις διευθύνσεις μνήμης που βρίσκονται οι virtual methods του αντικειμένου. Αυτό που κάνει ένας constructor είναι να δημιουργήσει μία σύνδεση μεταξύ της μεταβλητής μας (instance ενός object) και του VMT ώστε να μπορούν να κληθούν οι virtual μέθοδοι του αντικειμένου.

Όταν π.χ. η B.MoveTo θα πρέπει να καλέσει τη Hide, "πηγαίνει" στον VMT για να "δεί" τη διεύθυνση της Hide για το αντικείμενο B, έτσι ώστε να καλέσει τη σωστή Hide. Η διεύθυνση αυτή, όπως έχουμε ήδη αναφέρει, δεν είναι γνωστή την ώρα του compilation γιατί η Hide είναι virtual (μη πραγματική). Καταλαβαίνουμε λοιπόν ότι όταν πρέπει να κληθεί η Hide θα πρέπει να έχει γίνει ήδη η σύνδεση της μεταβλητής B με τον VMT, δηλαδή να έχει ήδη εκτελεστεί ο constructor της μεταβλητής B.

Μπορούμε να κάνουμε οποιαδήποτε μέθοδο ενός αντικειμένου constructor. Αρκεί να την καλούμε πρώτη, πριν απ' όλες τις άλλες. Στα objects που δώσαμε για παράδειγμα, κάναμε constructors τις μεθόδους με το όνομα Init, τις μεθόδους δηλαδή που μας χρησίμευαν για να δώσουμε τιμές στα πεδία των αντικειμένων μας. Αυτό είναι γενικά μία καλή συνήθεια. Εφόσον σχεδόν σε όλα τα objects υπάρχει μία ρουτίνα που δίνει αρχικές τιμές στα πεδία τους, καλό είναι εφόσον χρειαστούμε virtual methods, άρα και κάποιον constructor, να κάνουμε constructor τη ρουτίνα που φυσιολογικά θα καλούμε πρώτη.

Ενα object μπορεί να έχει πολλούς constructors. Για να λειτουργήσουν οι virtual methods αρκεί να κληθεί ένας απ' αυτούς. Ακόμα κι αν αργότερα καλέσουμε ξανά έναν constructor δεν υπάρχει κανένα πρόβλημα. Προσέξτε για παράδειγμα το παρακάτω αντικείμενο OneDate που κρατά μία ημερομηνία.

```
type
  OneDate = object
    Day,Month,Year : word;

    constructor Init ( nD,nM,nY : word );
    constructor InitToday;
    ....
end;
```

Εχουμε δύο τρόπους για να δώσουμε τιμές στα πεδία του. Την μέθοδο Init που δίνουμε τρεις νέες τιμές (μέρα, μήνας, χρόνος) και τη μέθοδο InitToday που παίρνει την ημερομηνία από το σύστημα. Εφόσον μας χρειάζεται ένας constructor μπορούμε πολύ απλά να κάνουμε και τις δύο μεθόδους constructors.

Επεκτασιμότητα των Objects

Το πιο σημαντικό πράγμα που πρέπει να σημειώσουμε για τα Units είναι ότι μπορούμε να δώσουμε (ή ακόμα και να πουλήσουμε) σε άλλους προγραμματιστές κάποια Units που περιέχουν objects που εμείς θεωρούμε χρήσιμα. Δίνουμε λοιπόν στους συναδέλφους μας έτοιμα μεταφρασμένα Units σε μορφή .TPU (ή .TPW για τα windows) χωρίς να είμαστε αναγκασμένοι να τους δώσουμε τον source κώδικα μας (το .PAS αρχείο δηλαδή). Το μόνο που χρειάζεται ίσως να δώσουμε είναι μία περιγραφή των αντικειμένων και των μεθόδων τους καθώς και ο τρόπος χειρισμού τους. Οποιοσδήποτε αποκτήσει τη βιβλιοθήκη μας λοιπόν μπορεί με την κληρονομικότητα και τον πολυμορφισμό του OOP να επεκτείνει τα έτοιμα δικά μας objects για δικό του όφελος.

Αυτή η πραγματικά χρήσιμη ιδιότητα, να παίρνεις τη δουλειά κάποιου άλλου (χωρίς τον αυθεντικό source κώδικά του) και να μπορείς να την επεκτείνεις, σύμφωνα με τις δικές σου ανάγκες και τα δικά σου κριτήρια χωρίς περιορισμούς, ονομάζεται **επεκτασιμότητα**. Η επεκτασιμότητα είναι η φυσική εξέλιξη της κληρονομικότητας. Κληρονομείς τα πάντα από τον πρόγονο σου και μετά προσθέτεις ότι νέα ιδιότητα μπορείς να φανταστείς. Οι virtual methods δίνουν και τη δυνατότητα να "σμίξουν" τα παλιά objects με τα καινούργια και έτσι η επέκταση των objects γίνεται πραγματικά παιχνιδάκι, ενώ το κόστος της είναι απλά ένας constructor και μία "επίσκεψη" στον VMT.

Συμβατότητα Objects

Με την εισαγωγή των objects και της κληρονομικότητας στην Pascal κάποιοι κανόνες συμβατότητας έχουν αλλάξει. Όπως είναι γνωστό η Pascal μπορεί να δεχθεί πρόταση assignment ($a := b;$) όταν και τα δύο μέρη της ισότητας είναι του ίδιου τύπου δεδομένων, ή ο τύπος δεδομένων δεξιά της ισότητας ορισμού ($:=$) είναι συμβατός με τον τύπο δεδομένων της μεταβλητής αριστερά (π.χ. integer με byte, string με char κ.λ.π.)

Στα objects μίας κληρονομικής σειράς μπορούμε να εξισώσουμε μία μεταβλητή (instance) οποιουδήποτε object με ένα instance του ίδιου object ή κάποιου απογόνου του. Το ίδιο ισχύει και για pointers σ' αυτά τα αντικείμενα. Έτσι αν έχουμε τις δηλώσεις :

```
type
  Location = object
    ...
  end;
  Pixel    = object (Location)
    ...
  end;
  Circle   = object (Pixel)
    ...
  end;

var
  L  : Location;
  P  : Pixel;
  C  : Circle;
  pL : ^Location;
  pP : ^Pixel;
  pC : ^Circle;
```

...οι παρακάτω προτάσεις είναι σωστές :

```
L := P;
L := C;
P := C;
pL := pP;
pL := pC;
pP := pC;
```

Οι αντίστροφες προτάσεις όμως είναι λάθος. Αυτό που συμβαίνει στις παραπάνω προτάσεις είναι απλώς εξίσωση των επιμέρους πεδίων των αντικειμένων που είναι κοινά και στα δύο αντικείμενα.

Επίσης, όταν έχουμε μία procedure ή function η οποία παίρνει παράμετρο κάποιο αντικείμενο κληρονομικής σειράς, τότε μπορούμε να "περάσουμε" σαν παράμετρο στη ρουτίνα ένα instance του αντικειμένου αυτού ή instance κάποιου απογόνου του. Αν δηλαδή έχουμε τη δήλωση :


```
procedure OneProc ( x : Location );
```

...οι παρακάτω προτάσεις είναι σωστές :

```
OneProc ( L );  
OneProc ( P );  
OneProc ( C );
```

Στην procedure θα περάσουν μόνο τα πεδία εκείνα που είναι κοινά με το αντικείμενο της δήλωσης της παραμέτρου.

ΚΕΦΑΛΑΙΟ 4

Δυναμικά Objects

Με τον όρο δυναμικά objects εννοούμε, pointers σε objects. Αυτό δεν θα πρέπει να προξενεί κανένα πρόβλημα. Οι pointers στα objects λειτουργούν ακριβώς όπως και στα records. Η σύνταξη της Pascal παραμένει ίδια. Στο παρακάτω παράδειγμα θα χρησιμοποιήσουμε το αντικείμενο TPixel και τον τύπο PPixel που είναι pointer σ' αυτό. Αν έχουμε λοιπόν τις δηλώσεις :

```
type
  PPixel = ^TPixel;
  TPixel = object (Location)
    Color : byte;

    constructor Init ( NewX,NewY : integer; NewC : byte );

    procedure Show; virtual;
    procedure Hide; virtual;
    procedure MoveTo ( NewX,NewY : integer );
  end;

var
  APtr : PPixel;
```

Οι παρακάτω προτάσεις είναι σωστές :

```
New (APtr);
APtr^.Init (100,150,2);
APtr^.X := 12;           { αλλάζω το πεδίο X }
APtr^.Show;
```

Όπως φαίνεται η σύνταξη της Pascal δεν αλλάζει είτε χρησιμοποιούμε κάποιο πεδίου του αντικειμένου (ακριβώς όπως στα records), είτε καλούμε μία μέθοδο του.

Allocation και Initialization με τη New

Στην Borland Pascal έγινε μία επέκταση της procedure New, ειδικά για τα δυναμικά objects, όπου το allocation και η κλήση του constructor (ή της συνηθισμένης ρουτίνας που καλούμε για να δώσουμε τιμές στο αντικείμενο) γίνονται σε μία γραμμή. Έτσι η New δέχεται δύο παραμέτρους. Η πρώτη είναι το όνομα της δυναμικής μεταβλητής (pointer) στο object και η δεύτερη είναι η κλήση του constructor του object. Το προηγούμενο παράδειγμα λοιπόν θα μπορούσε να είναι γίνει και με τον παρακάτω τρόπο :

```
New (APtr,Init (100,150,2));
```

Η New ακόμα επεκτάθηκε και σε function. Σαν function η New επιστρέφει έναν pointer του τύπου δεδομένων που της δίνουμε. Έτσι η πρόταση...

```
New (APtr);
```

...μπορεί να γραφεί...

```
APtr := New (PPixel);
```

Σημειώστε ότι το παραπάνω ισχύει για όλους τους pointers και όχι μόνο για pointers σε objects.

Όσον αφορά όμως και τα objects, η επέκταση της New σαν function μας επιτρέπει να γράψουμε το παρακάτω :

```
APtr := New (PPixel, Init (100,150,2));
```

Όπως φαίνεται, η πρώτη παράμετρος είναι ο τύπος δεδομένων που είναι pointer σε κάποιο αντικείμενο, και η δεύτερη παράμετρος είναι η κλήση του constructor του αντικείμενου αυτού.

Dispose για Δυναμικά Objects

Όπως και στην παραδοσιακή Pascal με τα records, τα αντικείμενα που έχουν γίνει allocated στο Heap, μπορούν να γίνουν deallocated με την Dispose όταν πλέον δεν τα χρειαζόμαστε :

```
Dispose (APtr);
```

Υπάρχουν όμως περιπτώσεις που το να "ξεφορτωθούμε" ένα αντικείμενο δεν σημαίνει απλά να κάνουμε deallocation τη μνήμη που κρατάει στο Heap. Ένα αντικείμενο μπορεί να περιέχει και άλλους pointers σε πίνακες, λίστες ή και άλλα αντικείμενα που θα πρέπει να "καθαριστούν" πριν γίνει deallocation το ίδιο το αντικείμενο. Οτιδήποτε είναι απαραίτητο να γίνει για να μπορέσουμε να ξεφορτωθούμε ένα αντικείμενο όταν πλέον δεν το χρειαζόμαστε, είναι καλό να συγκεντρωθεί σε μία μέθοδο (π.χ. Done) έτσι ώστε όταν την καλούμε να κάνει όλες αυτές τις δουλειές. Η μέθοδος αυτή πρέπει να περιέχει με κάθε λεπτομέρεια τις απαραίτητες κινήσεις για το καθάρισμα των pointers που περιέχει το αντικείμενο.

Destructors

Η Borland Pascal μας δίνει έναν ακόμη τύπο μεθόδων, τους Destructors (καταστροφείς) για να "καθαρίζουμε" και να επιστρέφουμε τη μνήμη που κρατούν τα δυναμικά αντικείμενα. Ένας destructor συνδυάζει το βήμα της επιστροφής της μνήμης (dispose) με οποιεσδήποτε άλλες ενέργειες απαιτούνται για να "καθαρίσουμε" ένα object. Όπως και με κάθε άλλη μέθοδο, μπορούμε να ορίσουμε πολλούς destructors για κάποιο object.

Ορίζουμε τους destructors μαζί με τις άλλες μεθόδους ενός αντικειμένου :

```
type
  TPixel = object (Location)
            Color : byte;

            constructor Init ( NewX,NewY : integer; NewC : byte );
            destructor Done; virtual;
            procedure Show; virtual;
            procedure Hide; virtual;
            procedure MoveTo ( NewX,NewY : integer );
          end;
```

Οι destructors μπορούν να κληρονομούνται και μπορούν να είναι static ή virtual. Επειδή γενικά πρέπει να εκτελούνται διαφορετικές εντολές για το "κλείσιμο" διαφορετικών objects, είναι μία καλή ιδέα να κάνετε πάντα τους destructors virtual, έτσι ώστε σε κάθε περίπτωση να καλείται ο σωστός destructor. Οι destructors δουλεύουν πραγματικά μόνο σε αντικείμενα που είναι δυναμικά. Δεν υπάρχει κανένα πρόβλημα να χρησιμοποιείτε destructors σε στατικά αντικείμενα. Απλά δεν γίνεται και τίποτε περισσότερο από την κλήση μίας μεθόδου.

Εκεί που οι destructors όμως είναι πραγματικά χρήσιμοι, είναι όταν έχουμε να κάνουμε με πολυμορφικά objects τα οποία πρέπει να καθαριστούν και να απελευθερωθεί ο χώρος από το Heap που έπιαναν.

Είναι γνωστό ότι μπορούμε να πούμε `APtr := BPtr` όπου `APtr` ένας pointer σε `TPixel` και `BPtr` ένας pointer σε `TCircle`, εφόσον το `TPixel` είναι πρόγονος του `TCircle`. Όμως τι θα γίνει αν κάνουμε `Dispose` το `APtr`; Θα γίνει deallocation της μνήμης που χρειάζεται ένα `TPixel` ή της μνήμης που χρειάζεται ένα `TCircle` (που είναι και το σωστό, αν υποθέσουμε ότι κάναμε `New` στο `BPtr`); Επειδή λοιπόν τα πολυμορφικά objects υποστηρίζουν αυτού του είδους τα "μπλεξίματα", η Pascal έρχεται να μας "ασφαλίσει" δίνοντας μας τους destructors και την καινούργια επέκταση της `Dispose`, η οποία (όπως και στη `New`) παίρνει σαν παραμέτρους το δυναμικό αντικείμενο και τον Destructor του. Έτσι λοιπόν γράφουμε :

```
Dispose (APtr,Done);
```

Αυτό που θα συμβεί εδώ, είναι ότι ο destructor του αντικειμένου που "δείχνει" ο APtr θα εκτελεστεί σαν μία κανονική μέθοδος. Σαν τελευταία όμως δουλειά, ο destructor, θα "πάει" στον VMT και θα "μάθει" πόσα bytes πιάνει αυτό το αντικείμενο, για να το περάσει στη συνέχεια στην Dispose. Έτσι η Dispose θα ολοκληρώσει το "κλείσιμο" του αντικειμένου που "έδειχνε" ο APtr, απελευθερώνοντας το σωστό αριθμό από bytes από το Heap.

ΠΡΟΣΟΧΗ. Ο destructor μόνος του δεν κάνει τη δουλειά του deallocation της μνήμης που πιάνει ένα object. Η δουλειά αυτή γίνεται **μόνο** από τη **Dispose**.

Σημειώστε επίσης ότι ο ίδιος ο destructor θα μπορούσε να ήταν μία **κενή** μέθοδος, απλά και μόνο για να εκμεταλευτούμε αυτή την ευκολία της "επίσκεψης" στον VMT :

```
destructor AnObject.Done;  
begin  
end;
```

Αυτό που κάνει τη χρήσιμη δουλειά στους destructors δεν είναι τόσο οι εντολές που περιέχουν (αυτές θα μπορούσαν να βρίσκονται σε οποιαδήποτε άλλη μέθοδο), αλλά οι ειδικές εντολές που προσθέτει ο compiler στο τέλος τους, ο **επίλογος** των destructors.

Και τώρα, τι κάνουμε;

Όπως με κάθε πλευρά του προγραμματισμού, έτσι και με τον Object Oriented Προγραμματισμό, δεν θα βελτιωθείτε διαβάζοντας ένα βιβλίο. Θα βελτιωθείτε και θα μάθετε πραγματικά, όταν το πιάσετε στα χέρια σας, όταν δουλέψετε. Ο περισσότερος κόσμος στην πρώτη του επαφή με τον OOP απαντά με ένα "Δεν κατάλαβα!" ή "Δεν είναι για μένα". Το πρώτο επιφώνημα χαράς όμως έρχεται μόλις αρχίσουν να τοποθετούν κάποια αντικείμενα στο πρόγραμμά τους και καταλάβουν το encapsulation. Το encapsulation όμως δεν είναι ο OOP. Είναι απλά η αρχή της όλης ιστορίας.

Μόλις το καταφέρετε, αρχίστε να εισάγετε τα objects σε κάθε καθημερινό σας πρόγραμμα. Σιγά σιγά θα αρχίσετε να μπαίνετε και στο νόημα της κληρονομικότητας. Συνήθως μόλις ανακαλύψετε ότι πάτε να φτιάξετε κάποιο αντικείμενο που μοιάζει πάρα πολύ με κάποιο άλλο, που μόλις πριν λίγες μέρες φτιάξατε.

Ο πολυμορφισμός όπως εξηγήσαμε και στο αντίστοιχο κεφάλαιο, προέρχεται από ένα πρόβλημα που έχουμε στις κληρονομικές σειρές. Είναι συγκεκριμένος. Έχει προϋποθέσεις. Αν δεν ισχύουν οι προϋποθέσεις αυτές, δεν θα χρειαστεί να ασχοληθείτε καθόλου. Όμως, ο πολυμορφισμός είναι η πραγματική δύναμη του OOP. Είναι το τελειότερο στάδιο τεχνικής προγραμματισμού σήμερα. Την ημέρα που θα τον έχετε καταλάβει πραγματικά και θα τον έχετε χρησιμοποιήσει αποτελεσματικά θα είστε ένας επιτυχημένος γνώστης του OOP.

Ρίξτε μία ματιά στις βιβλιοθήκες που έχετε φτιάξει σε παραδοσιακή Pascal. Προσπαθείστε να ανακαλύψετε τα αντικείμενα που κρύβονται εκεί. Μη διστάσετε να ξαναγράψετε από την αρχή τις βιβλιοθήκες σας αυτές, με την ιδέα του OOP. Γρήγορα θα καταλάβετε ότι όση δουλειά κι αν κάνετε, δεν θα πάει χαμένη. Η επαναχρησιμότητα (reusability) των Object Oriented Libraries είναι μεγαλύτερη από οτιδήποτε άλλο έχετε δοκιμάσει μέχρι στιγμής. Σχεδόν ποτέ δε θα χρειαστεί να ξαναγράψετε ένα object από την αρχή. Αν σας βολεύει όπως είναι, χρησιμοποιήστε το. Αν σας λείπει κάτι, συμπληρώστε το. Μην ξεχνάτε όμως : Ένα object που δουλεύει, δεν είναι ποτέ άχρηστο.

ΠΑΡΑΡΤΗΜΑ Α

Library Reference

Στο κεφάλαιο αυτό θα δούμε τη σύνταξη και χρήση ορισμένων από τις πιο χρήσιμες ρουτίνες (procedures & functions), μεταβλητές, σταθερές και types που περιέχονται στα έτοιμα units της Turbo Pascal. Η ταξινόμηση έγινε κατά Unit για μεγαλύτερη ευκολία.

System Unit

Το System Unit είναι ένα unit της Turbo Pascal το οποίο περιέχεται σε κάθε πρόγραμμά μας χωρίς να είμαστε υποχρεωμένοι να το συμπεριλάβουμε στα uses.

- **function Abs (x : integer ή real type) : ίδιο με το x ;**

Επιστρέφει την απόλυτη τιμή του x. Όπως φαίνεται το x μπορεί να είναι οποιοδήποτε integer type ή real type. Π.χ.

```
r := Abs(-2.3);
```

- **function ArcTan (x : real) : real;**

Επιστρέφει το τόξο εφαπτομένης του x, σε ακτίνια. Το τόξο εφαπτομένης είναι η γωνία που έχει εφαπτομένη x.

- **procedure Break;**

"Βγαίνει" από το loop στο οποίο βρισκόμαστε (for, while ή repeat). Είναι αντίστοιχη με ένα goto στην πρόταση μετά το loop. Η break είναι "δανεισμένη" από τη γλώσσα C και η χρήση της θα πρέπει να είναι σχετικά περιορισμένη. Π.χ.

```
for i := 1 to 15 do
begin
  readln (A[i]);
  if A[i] = 0 then
    break;
  writeln (A[i]);
end;
```

- **procedure ChDir (s : string);**

Πηγαίνει στο directory που περιγράφει το s. Αν στο s συμπεριλαμβάνεται και το όνομα ενός drive (π.χ. 'D:\WINDOWS\') αλλάζει και το drive. Αντίστοιχη με την εντολή **cd** του DOS.

- **procedure Continue;**

Δανεισμένη κι αυτή από τη γλώσσα C, όπως και η Break, αναγκάζει το τρέχον loop (for, while ή repeat) να αποφύγει όλες τις υπόλοιπες εντολές και να συνεχίσει ξανά με την επόμενη επανάληψη. Π.χ.

```
for i := 1 to 10 do
begin
  readln (A[i]);
  if A[i] = 0 then
    continue;
  writeln (A[i]);
end;
```

- **function Cos (x : real) : real;**

Επιστρέφει το ημίτονο της γωνίας x. Το x πρέπει να είναι εκφρασμένο σε ακτίνια (π ακτίνια = 180 μοίρες).

- **procedure Dec (var x : ordinal type [; n : longint]);**

Μειώνει την τιμή μίας μεταβλητής κατά 1. Αν δώσουμε τιμή στο n (δεύτερη προαιρετική παράμετρος) τότε η τιμή του x μειώνεται κατά n. Δουλεύει με όλα τα ordinal types και είναι ιδιαίτερα χρήσιμη στα Chars. Η **Dec(x)** είναι αντίστοιχη του καθιερωμένου "x := x - 1;" ενώ η **Dec(x,n)** είναι αντίστοιχη με "x := x-n;".

- **procedure Erase (var f);**

Σβήνει το αρχείο f. Προηγουμένως θα πρέπει να έχει γίνει Assign το f με κάποιο όνομα αρχείου του Dos. Αντίστοιχη με την εντολή **del** του DOS. Ποτέ μη χρησιμοποιήτε την Erase σε ανοικτό αρχείο.

- **procedure Exit;**

Φεύγει αμέσως από την τρέχουσα procedure ή function. Αν εκτελεστεί μέσα στο κυρίως πρόγραμμα τότε φεύγει αμέσως από το πρόγραμμα. Π.χ.

```
procedure WasteTime;
begin
  repeat
    if KeyPressed then
      Exit;
    write ('.');
  until false;
end;
```


- **function Exp (x : real) : real;**

Επιστρέφει το e υψωμένο στη δύναμη x . Το e είναι η βάση των φυσικών λογαρίθμων. Επειδή η Pascal δεν έχει ύψωση σε δύναμη χρησιμοποιείται σε συνδυασμό με τη συνάρτηση Ln και με ιδιότητες των λογαρίθμων για να υψώσουμε κάποιον αριθμό a στη δύναμη b , ως εξής :

```
function PowerOf ( a,b : real );
begin
    PowerOf := Exp(b*Ln(a));
end;
```

- **procedure FillChar (var x; Count : word; Value : _);**

Γεμίζει πολύ γρήγορα μία περιοχή της μνήμης (από τη θέση της μεταβλητής x) και μεγέθους Count bytes με την τιμή Value. Επειδή η FillChar δεν κάνει καθόλου ελέγχους, είναι πολύ επικίνδυνη και πρέπει να χρησιμοποιείται με προσοχή. Είναι ιδιαίτερα χρήσιμη για να γεμίζουμε μεγάλους πίνακες με μηδενικά. Καλό είναι να καλούμε τη συνάρτηση SizeOf για να παίρνουμε το μέγεθος μίας μεταβλητής (πίνακα - record) σε bytes. Π.χ.

```
var
    A : array[1..12000] of byte;
begin
    FillChar (A, SizeOf(A), 0);
end.
```

- **function Frac (x : real) : real;**

Επιστρέφει το δεκαδικό μέρος του αριθμού x . Ουσιαστικά $Frac(x) := x - Int(x)$;

- **procedure GetDir (d : byte; var s : string);**

Επιστρέφει στη μεταβλητή string s , το όνομα του τρέχοντος directory σε full path του Drive d . Αν $d = 0$ εννοείται το τρέχον drive, $d = 1$ είναι το Drive A:, 2 είναι το Drive B: κ.ο.κ.

- **procedure Halt [(ExitCode : word)];**

Σταματάει αμέσως την εκτέλεση του προγράμματος. Προαιρετικά μπορούμε να θέσουμε και έναν αριθμό στο ExitCode. Ο αριθμός αυτός επιστρέφεται στο Dos για χρήση από Batch Files. Αν δεν δώσουμε ExitCode, η Halt ισοδυναμεί με Halt (0);

- **procedure Inc (var x : ordinal type [; n : longint]);**

Αυξάνει την τιμή μίας μεταβλητής κατά 1. Αν δώσουμε τιμή στο n (δεύτερη προαιρετική παράμετρος) τότε η τιμή του x αυξάνεται κατά n . Είναι αντίστοιχη της Dec και λειτουργεί όπως ακριβώς κι αυτή.

- **function Int (x : real) : real;**

Επιστρέφει το ακέραιο μέρος του αριθμού x. Πρέπει να προσέξουμε ότι το αποτέλεσμα είναι πάλι real αριθμός.

```
r := Int (123.456); { 123.0 }
r := Int (-123.456); { -123.0 }
```

- **function IOResult : integer;**

Επιστρέφει το αποτέλεσμα της τελευταίας I/O λειτουργίας. Αν το IOResult είναι ίσο με 0, σημαίνει ότι η τελευταία λειτουργία I/O που έγινε ήταν επιτυχής. Σε οποιαδήποτε άλλη περίπτωση, επιστρέφεται ένας άλλος αριθμός. Αυτά ισχύουν όταν το I/O Checking είναι Off. Όταν είναι On ξέρουμε ότι σε κάθε λάθος I/O η Pascal δημιουργεί ένα Run-Time Error και διακόπτει την εκτέλεση του προγράμματος.

- **function Ln (x : real) : real;**

Επιστρέφει το φυσικό λογάριθμο του x. (Δείτε και την Exp).

- **procedure Mkdir (s : string);**

Δημιουργεί το directory που περιγράφεται στο string s. Αντίστοιχη με την εντολή **md** του DOS.

- **procedure Move (var Source, Dest; Count : word);**

Αντιγράφει Count bytes, από τη διεύθυνση μνήμης που ξεκινά η μεταβλητή Source, στη διεύθυνση μνήμης που ξεκινά η μεταβλητή Dest. Είναι πολύ γρήγορη, αλλά όπως και η FillChar επικίνδυνη. Πρέπει να χρησιμοποιείται με ιδιαίτερη προσοχή.

- **function Odd (x : longint) : boolean;**

Επιστρέφει true αν ο x είναι περιττός (μονός) αριθμός και false αν είναι άρτιος (ζυγός). Το ίδιο μπορεί να ελεγχθεί και με mod 2 βέβαια.

- **function Ord (x : ordinal type) : longint;**

Επιστρέφει τον αριθμό σειράς του x, όπου x μία έκφραση ordinal τύπου. Όταν ο x είναι χαρακτήρας τότε επιστρέφεται ο κωδικός του ASCII.

- **function ParamCount : word;**

Επιστρέφει τον αριθμό των παραμέτρων της command line. Σε συνδυασμό με την ParamStr μπορούμε να πάρουμε όλες τις παραμέτρους που έγραψε ο χρήστης στην command line (γραμμή εκτέλεσης του προγράμματος μας από το DOS).

- **function ParamStr (Index : byte) : string;**

Επιστρέφει το περιεχόμενο της παραμέτρου με αριθμό Index. Όταν το Index είναι 0, τότε επιστρέφει το full path του προγράμματος μας (π.χ. C:\UTILITY\L.EXE). Έτσι μπορούμε να μάθουμε και σε πιο directory είναι το πρόγραμμά μας αλλά και πιο είναι το όνομα του. Π.χ.

```
var
  i : byte;
begin
  writeln ('Parameters are : ');
  for i := 0 to ParamCount do
    writeln (i:3, ' ', ParamStr(i));
  end.
```

- **function Pi :real;**

Επιστρέφει την τιμή του αριθμού $\pi = 3.1415926535897932385$. Η ακρίβεια διαφέρει ανάλογα με τον τύπο δεδομένων που χρησιμοποιούμε (real, extended, double κ.λ.π.)

- **function Pred (x : ordinal type) : ίδιο type με του x ;**

Επιστρέφει τον προηγούμενο του x. Λειτουργεί με ordinal types, που σημαίνει ότι υποστηρίζει και chars και enumerated types. (βλ. Succ);

- **function Random : real;**

- **function Random (n : word) : word;**

Επιστρέφει έναν τυχαίο πραγματικό αριθμό x. ($0 \leq x < 1$). Αν χρησιμοποιήσουμε παράμετρο (n) τότε επιστρέφει έναν φυσικό αριθμό από 0 έως n-1. Προσέξτε ότι το n είναι ουσιαστικά το πλήθος των τυχαίων που θέλουμε. Έτσι για ένα ζάρι που έχει 6 τυχαίες περιπτώσεις θα χρειαστούμε Random(6). Επειδή όμως η Random(6) θα μας δώσει αριθμούς από 0 έως 5, προσθέτουμε το 1 και παίρνουμε αριθμούς από το 1 έως το 6. Δηλαδή :

```
begin
  for i := 1 to 10 do
    writeln ('Το ζάρι έφερε : ', Random(6)+1);
  end.
```

- **procedure Randomize;**

Αλλάζει την αρχική τιμή της γεννήτριας τυχαίων αριθμών, έτσι ώστε κάθε φορά που τρέχουμε ένα πρόγραμμα, να μην μας δίνει η Random τους ίδιους τυχαίους αριθμούς. Η κλήση στην Randomize αρκεί να γίνει μία μόνο φορά σε κάθε πρόγραμμα (συνήθως στην αρχή του προγράμματος).

- **procedure Rename (var f; Newname : string);**

Αλλάζει το όνομα του αρχείου f. Το αρχείο f πρέπει να το έχουμε κάνει Assign σε κάποιο όνομα αρχείου του DOS που υπάρχει. Αντίστοιχη με την εντολή **ren** του DOS. Ποτέ μην χρησιμοποιείτε Rename σε ανοικτά αρχεία.

- **procedure RmDir (s : string);**

Σβήνει το directory που περιγράφεται στο string s. Το directory πρέπει να υπάρχει, να μην είναι το τρέχον και να μην περιέχει αρχεία. Αντίστοιχη με την εντολή **rd** του DOS.

- **function Round (x : real) : longint;**

Στρογγυλοποιεί έναν πραγματικό αριθμό στον κοντινότερο ακέραιο. Προσέξτε ότι επιστρέφει longint και γι' αυτό δημιουργείται λάθος, αν το αποτέλεσμα δεν ανήκει στην περιοχή τιμών του τύπου longint.

```
i := Round (3.14);    { 3 }
i := Round (3.88);    { 4 }
```

- **function Sin (x : real) : real;**

Επιστρέφει το ημίτονο της γωνίας x. . Το x πρέπει να είναι εκφρασμένο σε ακτίνια (π ακτίνια = 180 μοίρες).

- **function SizeOf (x) : word;**

Επιστρέφει το μέγεθος που καταλαμβάνει το x στη μνήμη σε bytes. Το x μπορεί να είναι κάποια μεταβλητή ή το όνομα κάποιου τύπου δεδομένων. Π.χ.

```
var
  i : integer;
begin
  writeln ('Η μεταβλητή i πιάνει ',SizeOf(i),' bytes στη μνήμη. ');
  writeln ('Ένας longint πιάνει ',SizeOf(longint)' bytes στη μνήμη. ');
end.
```

- **function Sqr (x : integer ή real type) : ίδιου τύπου με το x ;**

Επιστρέφει το τετράγωνο του x (x^2).

- **function Sqrt (x : real) : real;**

Επιστρέφει την τετραγωνική ρίζα του x.

- **function Succ (x : ordinal type) : ίδιο type με του x ;**

Επιστρέφει τον επόμενο του x. Λειτουργεί όπως η Pred που επιστρέφει τον προηγούμενο του x. Το **x := Pred(x);** είναι ισοδύναμο με **Dec (x);**.

- **function Trunc (x : real) : longint;**

Μετατρέπει το x σε ακέραιο, αφαιρώντας τα δεκαδικά του ψηφία.

```
i := Trunc (3.14);    { 3 }  
i := Trunc (3.88);    { 3 }
```

- **function UpCase (c : char) : char;**

Μετατρέπει τον χαρακτήρα c σε κεφαλαίο. Λειτουργεί μόνο για το λατινικό αλφάβητο. Αν ο χαρακτήρας δεν είναι μεταξύ a..z, τον αφήνει ανέπαφο.

Τέλος στο System Unit συμπεριλαμβάνονται και οι ρουτίνες που έχουμε περιγράψει σε αντίστοιχα κεφάλαια :

Αρχεία: Append, Assign, BlockRead, BlockWrite, Close, Eof, Eoln, FilePos, FileSize, Read, ReadLn, Write, WriteLn, Reset, Rewrite, Seek.

Strings: Chr, Concat, Copy, Delete, Insert, Length, Pos, Str, Val.

Pointers: Dispose, FreeMem, GetMem, MaxAvail, MemAvail, New.

Crt Unit

Το Crt unit περιέχει μεταβλητές, σταθερές, procedures και functions χρήσιμες για το χειρισμό του πληκτρολογίου και της οθόνης.

- **var CheckBreak : boolean;**

Καθορίζει αν θα γίνεται έλεγχος για Ctrl + Break. Όταν η μεταβλητή αυτή είναι true η Pascal ελέγχει σε κάθε I/O λειτουργία αν πατήθηκε Ctrl+Break και διακόπτει την εκτέλεση του προγράμματος. Όταν το CheckBreak = false δεν γίνεται κανένας έλεγχος. Η CheckBreak είναι true εξ' ορισμού (by default). Σημειώστε ότι όταν τρέχουμε το πρόγραμμά μας κάτω από το περιβάλλον της Pascal, τότε το Ctrl+Break δουλεύει έτσι κι αλλιώς. Τα παραπάνω ισχύουν όταν το πρόγραμμα γίνει .EXE και τρέξει κάτω από DOS.

- **procedure ClrEol;**

Καθαρίζει όλους τους χαρακτήρες από τη θέση του cursor μέχρι το τέλος της γραμμής, χωρίς να μετακινεί τον cursor.

- **procedure ClrScr;**

Καθαρίζει την οθόνη και μετακινεί τον cursor στη θέση 1,1.

- **procedure Delay (ms : word);**

Καθυστερεί την εκτέλεση του προγράμματος για ms χιλιοστά του δευτερολέπτου (milliseconds). Η Delay είναι μία προσέγγιση, οπότε η περίοδος καθυστέρησης δεν είναι πάντα ακριβώς ms milliseconds.

- **procedure GotoXY (x,y : byte);**

Πηγαίνει το cursor στη στήλη x και στη γραμμή y της οθόνης.

- **function KeyPressed : boolean;**

Επιστρέφει true αν έχει πατηθεί πλήκτρο και false αν όχι. Η KeyPressed δεν μπορεί να καταλάβει αν πατήθηκαν πλήκτρα όπως Shift, Alt, CapsLock, Ctrl, NumLock κ.λ.π. Η KeyPressed όταν δώσει true, συνεχίζει να δίνει true επ' αόριστον, μέχρι να διαβαστεί το πλήκτρο αυτό είτε από μία ReadKey, είτε από Read ή ReadLn. Π.χ.

```
write ('Press any key');  
repeat  
until KeyPressed;
```

- **function ReadKey : char;**

Επιστρέφει τον χαρακτήρα που αντιστοιχεί στο πλήκτρο που πατήθηκε. Αν δεν έχει πατηθεί πλήκτρο τότε διακόπτει την εκτέλεση του προγράμματος μέχρι να πατηθεί κάποιο. Αν πατηθεί κάποιο από τα ειδικά πλήκτρα (βελάκια, function keys κ.λ.π.) τότε επιστρέφει τον χαρακτήρα #0. Στην περίπτωση αυτή πρέπει να ξαναζητήσουμε από τη ReadKey έναν χαρακτήρα (θα μας περιμένει) για να μπορέσουμε να καταλάβουμε ποιο πλήκτρο πατήθηκε.

```
case ReadKey of  
#27 : writeln ('Πάτησες το Esc');  
#32 : writeln ('Πάτησες το Space');  
#64 : writeln ('Πάτησες το @');  
#65 : writeln ('Πάτησες το A');
```

```
#0 : case ReadKey of
    #72 : writeln ('πάτησες το βελάκι πάνω');
    #61 : writeln ('πάτησες το F1');
    else writeln ('Δεν ξέρω τι πάτησες αλλά είναι ειδικό πλήκτρο');
    end;
else writeln ('Δεν ξέρω τι πάτησες, αλλά είναι κανονικό πλήκτρο');
end;
```

- **procedure TextBackground (Color : byte);**

Θέτει το χρώμα του background των χαρακτήρων. Το χρώμα μπορεί να είναι 0..7

- **procedure TextColor (Color : byte);**

Θέτει το χρώμα των χαρακτήρων. Το χρώμα μπορεί να είναι 0..15.

- **function WhereX : byte;**

- **function WhereY : byte;**

Επιστρέφουν τη θέση του cursor στην οθόνη. Η WhereX επιστρέφει τη στήλη και η WhereY τη γραμμή.

Dos Unit

Στο Dos Unit περιέχονται ρουτίνες που έχουν σχέση με το λειτουργικό σύστημα και τον υπολογιστή μας γενικότερα.

- **function DiskFree (Drive : byte) : longint;**

Επιστρέφει τον αριθμό των bytes που υπάρχουν ελεύθερα σε κάποια Drive. Αν Drive = 0 τότε θεωρείται το τρέχον drive. Drive = 1 είναι το Drive A:, 2 είναι το Drive B: κ.ο.κ. Η DiskFree επιστρέφει -1 αν το Drive δεν υπάρχει.

- **function DiskSize (Drive : byte) : longint;**

Λειτουργεί όπως ακριβώς και η DiskFree. Επιστρέφει τον συνολικό αριθμό των bytes που χωράει ένα drive.

- **procedure FindFirst (Path : string; Attr : word; var S : SearchRec);**

Ψάχνει για το πρώτο αρχείο σε κάποιο directory που ταιριάζει στη "μάσκα" που δίνουμε στο path και έχει τα attributes που ορίζουμε στο Attr. Το αποτέλεσμα του ψαξίματος επιστρέφεται στη μεταβλητή S τύπου SearchRec που περιγράφεται παρακάτω. Τα λάθη αναφέρονται στη μεταβλητή DosError. Αν η DosError γίνει 3 σημαίνει ότι δε βρέθηκε το path,

ενώ αν γίνει 18 σημαίνει ότι δεν υπάρχουν άλλα αρχεία. Σε συνδυασμό με την FindNext μπορεί να μας δώσει όλα τα αρχεία που έχουν τέτοιου είδους κοινά στοιχεία. Το Attr είναι μία από τις λέξεις : Archive, ReadOnly, System, Hidden, Directory, VolumeID ή συνδυασμός (με πρόσθεση (+)) αυτών ή η λέξη AnyFile που θα μας δώσει οτιδήποτε ταιριάζει με το path.

```
var
  DirInfo : SearchRec;
begin
  FindFirst ('*.*', AnyFile, DirInfo);
  while DosError = 0 do
    begin
      writeln (DirInfo.Name, ' ', DirInfo.Size);
      FindNext (DirInfo);
    end;
  end.
```

- **procedure FindNext (var S : SearchRec);**

Λειτουργεί σε συνδυασμό με την FindFirst και μας δίνει τα επόμενα αρχεία που μοιάζουν με τις προϋποθέσεις της FindFirst. (βλ. παράδειγμα παραπάνω).

- **procedure GetDate (var Year, Month, Day, DayOfWeek : word);**

Επιστρέφει την τρέχουσα ημερομηνία του συστήματος και την τοποθετεί στις τέσσερις μεταβλητές τύπου word που της δίνουμε. Το DayOfWeek είναι ένας αριθμός 0..6, όπου το 0 αντιστοιχεί στην Κυριακή.

- **procedure GetTime (var Hour, Minute, Second, Sec100 : word);**

Επιστρέφει την τρέχουσα ώρα του συστήματος στις τέσσερις μεταβλητές τύπου word που της δίνουμε. Το Sec100 είναι τα εκατοστά του δευτερολέπτου.

- **record SearchRec**

Το SearchRec είναι ο ορισμός ενός record που χρησιμοποιείται από τις FindFirst και FindNext για να μας δώσουν λεπτομέρειες για τα αρχεία που βρίσκονται στα directories. Ο ορισμός του είναι ως εξής :

```
type
  SearchRec = record
    Fill   : array[1..21] of byte;
    Attr   : byte;
    Time   : Longint;
    Size   : Longint;
    Name   : string[12];
  end;
```


Η ώρα και ημερομηνία είναι κωδικοποιημένες σε ένα Longint και χρειαζόμαστε την UnpackTime για να τις μετατρέψουμε σε αληθινές τιμές. Το Name περιέχει το όνομα του αρχείου μαζί με το extension και την τελεία (.) γι' αυτό είναι και δώδεκα χαρακτήρες.

- **procedure SetDate (Year, Month, Day : word);**

Θέτει νέα ημερομηνία στο σύστημα. Δεκτά έτη είναι 1980..2099, Μήνες 1..12 και ημέρες 1..31. Αν η ημερομηνία δεν υπάρχει (π.χ. 30 Φεβρουαρίου) τότε δεν γίνεται αλλαγή της ημερομηνίας του συστήματος.

- **procedure SetTime (Hour, Minute, Second, Sec100 : word);**

Θέτει νέα ώρα στο σύστημα. Το Hour είναι 0..23, το Minute 0..59, το Second 0..59 και το Sec100 (εκατοστά του δευτερολέπτου) 0..99. Αν η ώρα είναι λάθος, τότε δεν γίνεται καμία αλλαγή.

- **procedure UnpackTime (Time : longint; var DT : DateTime);**

Μετατρέπει έναν longint που περιέχει την ώρα και ημερομηνία ενός αρχείου του DOS σε ένα record τύπου DateTime που ορίζεται ως εξής :

```
type
  DateTime = record
    Year, Month, Day, Hour, Min, Sec : word;
  end;
```

ΠΑΡΑΡΤΗΜΑ Β

ASCII Table

Παρακάτω είναι ο πίνακας ASCII των υπολογιστών στο DOS με τους ελληνικούς χαρακτήρες (codepage #437).

ΠΑΡΑΡΤΗΜΑ Γ

Κωδικοί Πλήκτρων της ReadKey

Επιστρεφόμενοι κωδικοί πλήκτρων από τη συνάρτηση ReadKey για τα κανονικά πλήκτρα.

Πλήκτρο	Χαρακτήρας	Πλήκτρο	Χαρακτήρας
Ctrl-A .. Ctrl-Z	#1..#26	BackSpace	#8
Ctrl-[#27	Tab	#9
Ctrl-\	#28	Ctrl-Enter	#10
Ctrl-]	#29	Enter	#13
Ctrl-^	#30	Esc	#27
Ctrl-_	#31	Space	#32
#32..#255	#32..#255	Ctrl-BackSpace	#127

Όταν πατηθεί ειδικό πλήκτρο τότε η ReadKey επιστρέφει τον χαρακτήρα #0. Στην περίπτωση αυτή πρέπει να ξανακαλέσουμε τη ReadKey για να μας δώσει τον δεύτερο χαρακτήρα (που ήδη περιμένει) και είναι ο χαρακτήρας που θα μας δείξει το ειδικό πλήκτρο που πατήθηκε.

Πλήκτρο	Χαρακτήρας	Πλήκτρο	Χαρακτήρας
Ctrl-@	#3	F1..F10	#59..#68
Shift-Tab	#15	Shift-F1..F10	#84..#93
Alt-QWERTYUIOP	#16..#25	Ctrl-F1..F10	#94..#103
Alt-ASDFGHJKL	#30..#38	Alt-F1..F10	#104..#113
Alt-ZXCVBNM	#44..#50	Ctrl-PrtSc	#114
Home	#71	Ctrl-Left Arrow	#115
Up Arrow	#72	Ctrl-Right Arrow	#116
PgUp	#73	Ctrl-End	#117
Left Arrow	#75	Ctrl-PgDn	#118
Right Arrow	#77	Ctrl-Home	#119
End	#79	Alt-1234567890-=	#120..#131
Down Arrow	#80	Ctrl-PgUp	#132
PgDn	#81		
Ins	#82		
Del	#83		

ΠΑΡΑΡΤΗΜΑ Δ

Reserved Words

Στο παράρτημα αυτό θα βρείτε τις reserved words (δεσμευμένες λέξεις) και τα standard directives (τυποποιημένες οδηγίες) της Borland Pascal.

Οι reserved words έχουν το νόημα ότι δεν μπορούμε να τις χρησιμοποιήσουμε για δική μας χρήση (μεταβλητές, τύποι δεδομένων, κ.λ.π.)

and	exports	mod	shr
array	file	nil	string
asm	for	not	then
begin	function	object	to
case	goto	of	type
const	if	or	unit
constructor	implementation	packed	until
destructor	in	procedure	uses
div	inherited	program	var
do	inline	record	while
downto	interface	repeat	with
else	label	set	xor
end	library	shl	

Τα standard directives αντίθετα με τα reserved words μπορούν να ξαναορισθούν από εμάς. Βέβαια είναι καλό να αποφεύγεται η χρήση τους για ονόματα μεταβλητών κ.λ.π. γιατί μπορεί να προκαλέσει απροσδόκητα αποτελέσματα και να δυσκολέψει το debugging των προγραμμάτων μας.

absolute	far	name	resident
assembler	forward	near	virtual
export	index	private	
external	interrupt	public	

Οι λέξεις **private** και **public** είναι reserved words όταν βρίσκονται μέσα στις δηλώσεις objects.

ΠΑΡΑΡΤΗΜΑ Ε

Κανόνες Γραφής της Pascal

Οι παρακάτω κανόνες, δεν βρίσκονται εδώ για να δυσκολέψουν τη ζωή μας, αλλά για να προσπαθήσουν να καλυτερέψουν τον τρόπο με τον οποίο γράφουμε. Τα προγράμματά μας πρέπει να είναι όμορφα γραμμένα, ευανάγνωστα, με κενά που χωρίζουν το νόημα των προτάσεων που γράφουμε κ.λ.π. Να θυμάστε πάντα ότι ο source κώδικας είναι δικός σας και κανένας δεν θα χρειαστεί ίσως ποτέ να τον δει και να τον διαβάσει. Εσείς και μόνο εσείς. Υιοθετήστε λοιπόν ένα στυλ γραψίματος, το οποίο θα σας βοηθά να διαβάζετε εύκολα και γρήγορα τον κώδικά σας, όταν ψάχνετε για τυχόν λάθη. Δεν είναι λίγοι εκείνοι οι οποίοι έχουν μονολογήσει : "Θεέ μου, δημιούργησα ένα τέρας!" εννοώντας το πρόγραμμά τους στο οποίο δεν καταλαβαίνουν τί έχουν κάνει.

Πρέπει λοιπόν πάνω απ' όλα να καταλάβετε ότι τα προγράμματα, τα γράφουμε σήμερα, αλλά μπορεί να τα ξαναχρειαστούμε μετά από πολλούς μήνες, ή χρόνια. Επίσης, πρέπει να αντιληφθείτε ότι ένα πρόγραμμα μεγάλο σε μέγεθος (πάνω από 1000 γραμμές π.χ.) είναι πολύ δύσκολο να το χειριστείτε σε έναν editor μετακινούμενοι πάνω - κάτω συνέχεια. Η δόμηση λοιπόν του προγράμματος (χωρισμός σε procedures, OOP κ.λ.π.) καθώς και η όμορφη γραφή του κώδικα είναι σημεία πολύ σημαντικά στη δημιουργία προγραμμάτων και προγραμματιστών.

Παρακάτω υπάρχουν ορισμένοι κανόνες γραφής. Είτε τους υιοθετήσετε πλήρως, είτε με μερικές παραλλαγές να ξέρετε ότι βαδίζετε στο σωστό δρόμο. Στο δρόμο που βαδίζουν έμπειροι προγραμματιστές εδώ και χρόνια, άνθρωποι που έχουν γράψει δεκάδες ή εκατοντάδες χιλιάδες γραμμές κώδικα...

- Βάζετε αρχικά σχόλια στο πρόγραμμά σας. Μερικές γραμμές με το όνομα του προγράμματος, την ημερομηνία που το ξεκινήσατε, την ημερομηνία τελευταίας αλλαγής κ.λ.π. είναι πολύ χρήσιμες μετά από αρκετό καιρό.

```
{ Flight Simulator v5.0                      }
{ a very nice program written by Me          }
{                                     Started : 12.May.1992 }
{                                     Last update : 12.May.1994 }
```

- Οι λέξεις `uses`, `const`, `type` και `var` των δηλώσεων πρέπει να βρίσκονται μόνες τους σε μία γραμμή και η προηγούμενη γραμμή να έχει μείνει κενή.

- Κάθε πρόταση να είναι γραμμένη σε μία γραμμή μόνη της. Αντί :

```
for i := 1 to 10 do writeln (i);
if i > 5 then x := y;
a := 3; b := 5; c := 7;
```

...καλύτερο είναι να γράφουμε :

```
for i := 1 to 10 do
    writeln (i);
if i > 5 then
    x := y;
a := 3;
b := 5;
c := 7;
```

- Μεταξύ των `procedures` & `functions` να υπάρχει τουλάχιστον μία κενή γραμμή.
- Πριν και μετά τα `:=`, `=` και `:` να υπάρχει ένα κενό. Επίσης κενά βάζουμε πριν και μετά την παρένθεση (καθώς και πριν το `)`; στις επικεφαλίδες των `procedures`.

```
for i := 1 to 10 do
    ....

function One ( a : byte ) : integer;
```

- Τα σχόλια είναι καλό να είναι σε γραμμές μόνα τους και να μην μπλέκονται με τον κώδικα κάνοντάς τον πολλές φορές δύσκολο να διαβαστεί.
- Οι λέξεις `repeat`, `begin`, `record` να βρίσκονται μόνες τους σε μία γραμμή και να μην τις ακολουθεί κώδικας.
- Το `end` των `begin`, `case`, `record`, `object` να βρίσκεται στην ίδια στήλη με αυτές τις λέξεις. Είναι πολύ χρήσιμο, βλέποντας ένα `end` να πηγαίνουμε κάθετα προς τα επάνω και να βρίσκουμε την αρχή του `block`.

```
type
    a = record
        X,Y : integer;
    end;
```

- Χρησιμοποιείτε ονόματα μεταβλητών κατανοητά, που δείχνουν ακριβώς τί τιμές παίρνουν οι μεταβλητές αυτές.

- Όλες τις σταθερές των προγραμμάτων σας να τις δηλώνετε δίνοντάς τους χαρακτηριστικά ονόματα. Πάντα συμβαίνει να θέλουμε να αλλάξουμε καμιά πενήνταριά 18 σε 19...
- Η ευθυγράμμιση (alignment) και το indentation (πόσο δεξιά γράφουμε κάποιες προτάσεις) των προγραμμάτων μας είναι πράγματα πολύ σημαντικά που ανεβάζουν το επίπεδο αναγνωσιμότητας των προγραμμάτων μας. Επίσης η χρήση κενών γραμμών είναι και αυτή πολύ βοηθητική. Χρησιμοποιείτε 3 ή 4 κενά (2 είναι λίγα, 5 είναι πολλά).
- Το εσωτερικό των while, repeat, for, if - then - else πρέπει να είναι γραμμένο δεξιότερα από την αρχική πρόταση. Όταν χρησιμοποιούμε σύνθετη πρόταση (begin - end) στις while, for και if, το begin και το end να είναι γραμμένο το ίδιο δεξιά με την αρχική πρόταση. Αυτό είναι καλό διότι σε περίπτωση που θέλουμε να βγάλουμε ή να βάλουμε ένα begin - end σε μία τέτοια πρόταση είναι πολύ εύκολο. Προσθέτουμε ή αφαιρούμε γραμμές χωρίς να μεταφέρονται δεξιά ή αριστερά οι εντολές στο εσωτερικό :

```

while true do
    SomeProc (a,b);
for i := 1 to 12 do
begin
    readln (A[i]);
    Sum := Sum+A[i];
end;
if i = 0 then
    write ('Zero')
else
begin
    write ('Not Zero');
    writeln ('-');
end;

```

- Όταν γράφουμε procedures μέσα σε procedures πρέπει να τις γράφουμε πιο δεξιά :

```

procedure One;
var
    i : byte;

    procedure Two;
    begin
        writeln ('*****');
    end;

begin
    for i := 1 to 5 do
        writeln ('@');
        Two;
        writeln ('@');
    end;

```

- Απαγορεύεται αυστηρά η χρήση goto. (Δεν υπάρχει μέσα στο βιβλίο κιόλας...) Αν θελήσετε σε κάποια στιγμή να χρησιμοποιήσετε goto, σταματήστε αμέσως τον προγραμματισμό, ηρεμήστε και επανέλθετε μετά από αρκετή ώρα. Οτι γίνεται με goto μπορεί να γίνει και διαφορετικά και μάλιστα με πολύ πιο δομημένο τρόπο. (Όλα αυτά για αυτούς που προέρχονται από τη Basic)

- Η σκάλα των if (if ladder) γράφεται ως εξής :

```

if <boolean expression> then
    ....
else
    if <boolean expression> then
        ....
    else
        if <boolean expression> then
            ....
        else
            ....;

```

ή

```

if <boolean expression> then
begin
    ....
end
else
    if <boolean expression> then
        begin
            ....
        end
    else
        begin
            ....
        end;

```

- Στην case προσπαθείστε να ευθυγραμμίσετε τα : , τα begin με τα end; εφ' όσον έχετε σύνθετες προτάσεις, και την πρόταση στο else με τις υπόλοιπες :

```

case OneByte of
3   : writeln ('a');
25  : begin
        writeln ('a');
        writeln ('b');
    end;
200 : writeln ('c');
else : writeln ('other');
end;

```

- Γενικά πάντα να είστε σε θέση να απομακρύνετε (ή να προσθέσετε) τα begin - end χωρίς να χρειάζεται μετακίνηση των εντολών που περιείχαν (ή που θα περιέχουν) δεξιά - αριστερά.

BIBΛΙΟΓΡΑΦΙΑ

- Steve Wood, ***Using Turbo Pascal***. McGraw-Hill, 1988.
- Herbert Schildt, ***Advanced Turbo Pascal Programming and Techiques***. McGraw-Hill, 1986.
- Kris Jamsa & Steven Nameroff, ***Turbo Pascal Programmer's Library***. McGraw-Hill, 1987.
- Paul Cilwa, ***Borland Pascal 7 Insider***. Coriolis Group Book, 1993.
- ***Turbo Pascal v5.0 Programmer's Reference***. Borland International, 1988.
- ***Turbo Pascal v5.5 OOP Guide***. Borland International, 1988.
- ***Borland Pascal with Objects v7.0 User's Guide***. Borland International, 1992.
- ***Borland Pascal with Objects v7.0 Language Guide***. Borland International, 1992.
- ***Borland Pascal with Objects v7.0 Programmer's Reference***. Borland International, 1992.
- ***Object Windows Programming Guide***. Borland International, 1992.