Notes:

Chapter 2
BASIC LANGUAGE ELEMENTS

Basic Symbols

The basic vocabulary of TURBO Pascal consists of basic symbols divided into letters, digits, and special symbols:

Letters
A to Z, a to z, and _ (underscore)
Digits
0 1 2 3 4 5 6 7 8 9
Special symbols
+ - * / = ^ < > ( ) [ ] { } . , : ; ' # $

No distinction is made between upper and lower case letters. Certain operators and delimiters are formed using two special symbols:

Assignment operator: :=
Relational operators: <> <= >=
Subrange delimiter: . .
Brackets: (. and .) may be used instead of [and]
Comments: (* and *) may be used instead of { and }

Reserved Words

Reserved words are integral parts of TURBO Pascal. They cannot be redefined and must therefore not be used as user defined identifiers.

```
* absolute    * external      nil         * shl
  and           file          not         * shr
  array         forward     * overlay     * string
  begin         for           of            then
  case          function      or            type
  const         goto          packed        to
  div         * inline        procedure     until
  do            if            program       var
  downto        in            record        while
  else          label         repeat        with
  end           mod           set         * xor
```

Throughout this manual, reserved words are written in **boldface**. The asterisks indicate reserved words not defined in standard Pascal.

Standard Identifiers

TURBO Pascal defines a number standard identifiers of predefined types, constants, variables, procedures, and functions. Any of these identifiers may be redefined but it will mean the loss of the facility offered by that particular identifier and may lead to confusion. The following standard identifiers are therefore best left to their special purposes:

```
Addr          Delay         Length        Release
ArcTan        Delete        Ln            Rename
Assign        EOF           Lo            Reset
Aux           EOLN          LowVideo      Rewrite
AuxInPtr      Erase         Lst           Round
AuxOutPtr     Execute       LstOutPtr     Seek
BlockRead     Exit          Mark          Sin
BlockWrite    Exp           MaxInt        SizeOf
Boolean       False         Mem           SeekEof
BufLen        FilePos       MemAvail      SeekEoln
Byte          FileSize      Move          Sqr
Chain         FillChar      New           Sqrt
Char          Flush         NormVideo     Str
Chr           Frac          Odd           Succ
Close         GetMem        Ord           Swap
ClrEOL        GotoXY        Output        Text
ClrScr        Halt          Pi            Trm
Con           HeapPtr       Port          True
ConInPtr      Hi            Pos           Trunc
ConOutPtr     IOresult      Pred          UpCase
Concat        Input         Ptr           Usr
ConstPtr      InsLine       Random        UsrInPtr
Copy          Insert        Randomize     UsrOutPtr
Cos           Int           Read          Val
CrtExit       Integer       ReadLn        Write
CrtInit       Kbd           Real          WriteLn
DelLine       KeyPressed
```

Each TURBO Pascal implementation further contains a number of dedicated standard identifiers which are listed in chapters 20, 21, and 22.

Throughout this manual, all identifiers, including standard identifiers, are written in a combination of upper and lower case letters (see page 43). In the text (as opposed to program examples), they are furthermore printed in *italics*.

Delimiters

Language elements must be separated by at least one of the following delimiters: a blank, an end of line, or a comment.

Program Lines

The maximum length of a program line is 127 characters; any character beyond the 127th is ignored by the compiler. For this reason the TURBO editor allows only 127 characters on a line, but source code prepared with other editors may use longer lines. If such a text is read into the TURBO editor, line breaks will be automatically inserted, and a warning is issued.

Notes:

Chapter 3
STANDARD SCALAR TYPES

A data type defines the set of values a variable may assume. Every variable in a program must be associated with one and only one data type. Although data types in TURBO Pascal can be quite sophisticated, they are all built from simple (unstructured) types.

A simple type may either be defined by the programmer (it is then called a *declared scalar type*), or be one of the *standard scalar types*: **integer**, **real**, **boolean**, **char**, or **byte**. The following is a description of these five standard scalar types.

Integer

Integers are whole numbers; in TURBO Pascal they are limited to a range of −32768 through 32767. Integers occupy two bytes in memory.

Overflow of integer arithmetic operations is not detected. Notice in particular that partial results in integer expressions must be kept within the integer range. For instance, the expression 1000 * 100 / 50 will not yield 2000, as the multiplication causes an overflow.

Byte

The type *Byte* is a subrange of the type *Integer*, of the range 0..255. Bytes are therefore compatible with integers. Whenever a *Byte* value is expected, an *Integer* value may be specified instead and vice versa, **except** when passed as parameters. Furthermore, *Bytes* and *Integers* may be mixed in expressions and *Byte* variables may be assigned integer values. A variable of type *Byte* occupies one byte in memory.

Real

The range of **real** numbers is 1E − 38 through 1E + 38 with a mantissa of up to 11 significant digits. Reals occupy 6 bytes in memory.

Overflow during an arithmetic operation involving reals causes the program to halt, displaying an execution error. An underflow will cause a result of zero.

Although the type **real** is included here as a standard scalar type, the following differences between **reals** and other scalar types should be noticed:

1) The functions *Pred* and *Succ* cannot take real arguments.
2) Reals cannot be used in array indexing.
3) Reals cannot be used to define the base type of a set.
4) Reals cannot be used in controlling **for** and **case** statements.
5) Subranges of reals are not allowed.

Boolean

A boolean value can assume either of the logical truth values denoted by the standard identifiers *True* and *False*. These are defined such that *False* < *True*. A *Boolean* variable occupies one byte in memory.

Char

A *Char* value is one character in the ASCII character set. Characters are ordered according to their ASCII value, for example: 'A' < 'B'. The ordinal (ASCII) values of characters range from 0 to 255. A *Char* variable occupies one byte in memory.

Chapter 4
USER DEFINED LANGUAGE ELEMENTS

Identifiers

Identifiers are used to denote labels, constants, types, variables, procedures, and functions. An identifier consists of a letter or underscore followed by any combination of letters, digits, or underscores. An identifier is limited in length only by the line length of 127 characters, and all characters are significant.

Examples:

```
TURBO
square
persons_counted
BirthDate
3rdRoot          illegal, starts with a digit
Two Words        illegal, must not contain a space
```

As TURBO Pascal does not distinguish between upper and lower case letters, the use of mixed upper and lower case as in *BirthDate* has no functional meaning. It is nevertheless encouraged as it leads to more legible identifiers. *VeryLongIdentifier* is easier to read for the human reader than *VERYLONGIDENTIFIER*. This mixed mode will be used for all identifiers throughout this manual.

Numbers

Numbers are constants of integer type or of real type. Integer constants are whole numbers expressed in either decimal or hexadecimal notation. Hexadecimal constants are identified by being preceded by a dollarsign: `$ABC` is a hexadecimal constant. The decimal integer range is −32768 through 32767 and the hexadecimal integer range is $0000 through $FFFF.

Examples:

```
1
12345
-1
$123
$ABC
$123G          illegal, G is not a legal hexadecimal digit
1.2345         illegal as an integer, contains a decimal parts
```

The range of *Real* numbers is 1E-38 through 1E + 38 with a mantissa of up to 11 significant digits. Exponential notation may be used, with the letter *E* preceding the scale factor meaning "times ten to the power of". An integer constant is allowed anywhere a real constant is allowed. Separators are not allowed within numbers.

Examples:

```
1.0
1234.5678
-0.012
1E6
2E-5
-1.23456789012E+12
1                          legal, but it is not a real, it is an integer
```

Strings

A string constant is a sequence of characters enclosed in single quotes:

```
'This is a string constant '
```

A single quote may be contained in a string by writing two successive single quotes. Strings containing only a single character are of the standard type *char*. A string is compatible with an **array of** *Char* of the same length. All string constants are compatible with all **string** types.

Examples:

```
'TURBO'
'You''ll see'
''''
';'
''
```


As shown in example 2 and 3, a single quote within a string is written as two consecutive quotes. The four consecutive single quotes in example 3 thus constitute a string containing *one* quote.

The last example - the quotes enclosing no characters, denoting *the empty string* - is compatible only with **string** types.

Control Characters

TURBO Pascal also allows control characters to be embedded in strings. Two notations for control characters are supported:
1) The # symbol followed by an integer constant in the range 0..255 denotes a character of the corresponding ASCII value, and
2) the ^ symbol followed by a character, denotes the corresponding control character.

Examples:

```
#10        ASCII 10 decimal (Line Feed).
#$1B       ASCII 1B hex (Escape).
^G         Control-G (Bell).
^L         Control-L (Form Feed).
^[         Control-[ (Escape).
```

Sequences of control characters may be concatenated into strings by writing them *without separators* between the individual characters:

```
#13#10
#27^U#20
^G^G^G^G
```

The above strings contain two, three, and four characters, respectively. Control characters may also be mixed with text strings:

```
'Waiting for input! '^G^G^G' Please wake up'
#27'U '
'This is another line of text '^M^J
```

These three strings contain 37, 3, and 31 characters, respectively.

Comments

A comment may be inserted anywhere in the program where a delimiter is legal. It is delimited by the curly braces { and }, which may be replaced by the symbols (* and *).

Examples:

```
{This is a comment}
(* and so is this *)
```

Curly braces may not be nested within curly braces, and (*..*) may not be nested within (*..*). However, curly braces may nested within (*..*) and vise versa, thus allowing entire sections of source code to be commented away, even if they contain comments.

Compiler Directives

A number of features of the TURBO Pascal compiler are controlled through compiler directives. A compiler directive is introduced as a comment with a special syntax which means that whenever a comment is allowed in a program, a compiler directive is also allowed.

A compiler directive consists of an opening brace immediately followed by a dollar-sign immediately followed by one compiler directive letter or a list of compiler directive letters separated by commas. The syntax of the directive or directive list depends upon the directive(s) selected. A full description of each of the compiler directives follow in the relevant sections; and a summary of compiler directives is located in Appendix C. File inclusion is discussed in chapter 17.

Examples:

```
{$I-}
{$I INCLUDE.FIL}
{$R-,B+,V-}
(*$X-*)
```

Notice that no spaces are allowed before or after the dollar-sign.

Chapter 5
PROGRAM HEADING AND PROGRAM BLOCK

A Pascal program consists of a program heading followed by a program block. The program block is further divided into a declaration part, in which all objects local to the program are defined, and a statement part, which specifies the actions to be executed upon these objects. Each is described in detail in the following.

Program Heading

In TURBO Pascal, the program heading is purely optional and of no significance to the program. If present, it gives the program a name, and optionally lists the parameters through which the program communicates with the environment. The list consists of a sequence of identifiers enclosed in parentheses and separated by commas.

Examples:

```
program Circles;
program Accountant(Input,Output);
program Writer(Input,Printer);
```

Declaration Part

The declaration part of a block declares all identifiers to be used within the statement part of that block (and possibly other blocks within it). The declaration part is divided into five different sections:

1) Label declaration part
2) Constant definition part
3) Type definition part
4) Variable declaration part
5) Procedure and function declaration part

Whereas standard Pascal specifies that each section may only occur zero or one time, and only in the above order, TURBO Pascal allows each of these sections to occur any number of times in any order in the declaration part.

Label Declaration Part

Any statement in a program may be prefixed with a **label**, enabling direct branching to that statement by a **goto** statement. A label consists of a label name followed by a colon. Before use, the label must be declared in a label declaration part. The reserved word **label** heads this part, and it is followed by a list of label identifiers separated by commas and terminated by a semi-colon.

Example:

```
label 10, error, 999, Quit;
```

Whereas standard Pascal limits labels to numbers of no more than 4 digits, TURBO Pascal allows both numbers and identifiers to be used as labels.

Constant Definition Part

The constant definition part introduces identifiers as synonyms for constant values. The reserved word **const** heads the constant definition part, and is followed by a list of constant assignments separated by semi-colons. Each constant assignment consists of an identifier followed by an equal sign and a constant. Constants are either strings or numbers as defined on pages 43 and 44.

Example:

```
const
  Limit = 255;
  Max = 1024;
  PassWord = 'SESAM';
  CursHome = ^['V';
```

The following constants are predefined in TURBO Pascal which may be referenced without previous definition:

```
Name:       Type and value:
Pi          Real  (3.1415926536E+00).
False       Boolean (the truth value false).
True        Boolean (the truth value true).
Maxint      Integer (32767).
```

As described in chapter 13, a constant definition part may also define typed constants.

Type Definition Part

A data type in Pascal may be either directly described in the variable declaration part or referenced by a type identifier. Several standard type identifiers are provided, and the programmer may create his own types through the use of the type definition. The reserved word **type** heads the type definition part, and it is followed by one or more type assignments separated by semi-colons. Each type assignment consists of a type identifier followed by an equal sign and a type.

Example:

```
type
  Number = Integer;
  Day = (mon,tues,wed,thur,fri,sat,sun);
  List = array[1..10] of Real;
```

More examples of type definitions are found in subsequent sections.

Variable Declaration Part

Every variable occurring in a program must be declared before use. The declaration must textually precede any use of the variable so that the variable is 'known' to the compiler when it is used.

A variable declaration consists of the reserved word **var** followed by one or more identifier(s), separated by commas, each followed by a colon and a **type**. This creates a new variable of the specified type and associates it with the specified identifier.

The 'scope' of this identifier is the block in which it is defined, and any block within that block. Note, however, that any such block within another block may define *another* variable using the *same* identifier. This variable is said to be *local* to the block in which it is declared (and any blocks within *that block*), and the variable declared on the outer level (the *global* variable) becomes inaccessible.

Example:

```
var
  Result, Intermediate, SubTotal: Real;
  I, J, X, Y: Integer;
  Accepted, Valid: Boolean;
  Period: Day;
  Buffer: array[0..127] of Byte;
```

Procedure and Function Declaration Part

A procedure declaration serves to define a procedure within the current procedure or program (see page 131). A procedure is activated from a procedure statement (see page 56), and upon completion, program execution continues with the statement immediately following the calling statement.

A function declaration serves to define a program part which computes and returns a value (see page 137). A function is activated when its designator is met as part of an expression (see page 54).

Statement Part

The statement part is the last part of a block. It specifies the actions to be executed by the program. The statement part takes the form of a compound statement followed by a period or a semi-colon. A compound statement consists of the reserved word **begin**, followed by a list of statements separated by semicolons, terminated by the reserved word **end**.

Chapter 6
EXPRESSIONS

Expressions are algorithmic constructs specifying rules for the computation of values. They consist of operands: variables, constants, and function designators combined by means of operators as defined in the following.

This section describes how to form expressions from the standard scalar types *Integer*, *Real*, *Boolean*, and *Char*. Expressions containing declared scalar types, **string** types, and **set** types are described on pages 63, 67, and 86, respectively.

Operators

Operators fall into five categories, denoted by their order of precedence:

1) Unary minus (minus with one operand only).
2) **Not** operator.
3) Multiplying operators: *, /, **div**, **mod**, **and**, **shl**, and **shr**.
4) Adding operators: +, −, **or**, and **xor**.
5) Relational operators: =, <>, <, >, <=, >=, and **in**.

Sequences of operators of the same precedence are evaluated from left to right. Expressions within parentheses are evaluated first and independently of preceding or succeeding operators.

If both of the operands of the multiplying and adding operators are of type *Integer*, then the result is of type Integer. If one (or both) of the operands is of type *Real*, then the result is also of type *Real*.

Unary Minus

The unary minus denotes a negation of its operand which may be of *Real* or *Integer* types.

Not Operator

The **not** operator negates (inverses) the logical value of its Boolean operand:

```
not True        = False
not False       = True
```

TURBO Pascal also allows the **not** operator to be applied to an *Integer* operand, in which case bitwise negation takes place.

Examples:

```
not 0           = -1
not -15         = 14
not $2345       = $DCBA
```

Multiplying Operators

Operator	Operation	Operand type	Result type
*	multiplication	Real	Real
*	multiplication	Integer	Integer
*	multiplication	Real, Integer	Real
/	division	Real, Integer	Real
/	division	Integer	Real
/	division	Real	Real
div	Integer division	Integer	Integer
mod	modulus	Integer	Integer
and	arithmetic **and**	Integer	Integer
and	logical **and**	Boolean	Boolean
shl	shift left	Integer	Integer
shr	shift right	Integer	Integer

Examples:

```
12 * 34         = 408
123 / 4         = 30.75
123 div 4       = 30
12 mod 5        = 2
True and False  = False
12 and 22       = 4
2 shl 7         = 256
256 shr 7       = 2
```


Adding Operators

Operator	Operation	Operand type	Result type
+	addition	Real	Real
+	addition	Integer	Integer
+	addition	Real, Integer	Real
−	subtraction	Real	Real
−	subtraction	Integer	Integer
−	subtraction	Real, Integer	Real
or	arithmetic **or**	Integer	Integer
or	logical **or**	Boolean	Boolean
xor	arithmetic **xor**	Integer	Integer
xor	logical **xor**	Boolean	Boolean

Examples:

```
123+456         =579
456-123.0       =333.0
True or False   =True
12 or 22        =30
True xor False  =True
12 xor 22       =26
```

Relational Operators

The relational operators work on all standard scalar types: *Real*, *Integer*, *Boolean*, *Char*, and *Byte*. Operands of type *Integer*, *Real*, and *Byte* may be mixed. The type of the result is always Boolean, i.e. *True* or *False*.

```
=     equal to
<>    not equal to
>     greater than
<     less than
>=    greater than or equal to
<=    less than or equal to
```

Examples:

`a = b`	true if a is equal to b.
`a <> b`	true if a is not equal to b.
`a > b`	true if a is greater than b.
`a < b`	true if a is less than b.
`a >= b`	true if a is greater than or equal to b.
`a <= b`	true if a is less than or equal to b.

Function Designators

A function designator is a function identifier optionally followed by a parameter list, which is one or more variables or expressions separated by commas and enclosed in parentheses. The occurrence of a function designator causes the function with that name to be activated. If the function is not one of the pre-defined standard functions, it must be declared before activation.

Examples:

```
Round(PlotPos)
Writeln(Pi * (Sqr(R)))
(Max(X,Y) < 25) and (Z > Sqrt(X * Y))
Volume(Radius,Height)
```


Chapter 7
STATEMENTS

The statement part defines the action to be carried out by the program (or subprogram) as a sequence of *statements*; each specifying one part of the action. In this sense Pascal is a sequential programming language: statements are executed sequentially in time; never simultaneously. The statement part is enclosed by the reserved words **begin** and **end** and within it, statements are separated by semi-colons. Statements may be either *simple* or *structured*.

Simple Statements

Simple statements are statements which contain no other statements. These are the assignment statement, procedure statement, **goto** statement, and empty statement.

Assignment Statement

The most fundamental of all statements is the assignment statement. It is used to specify that a certain value is to be assigned to a certain variable. An assignment consists of a variable identifier followed by the assignment operator **:=** followed by an expression.

Assignment is possible to variables of any type (except files) as long as the variable (or the function) and the expression are of the same type. As an exception, if the variable is of type *Real*, the type of the expression may be *Integer*.

Examples:

```
Angle := Angle * Pi;
AccessOK := False;
Entry := Answer = PassWord;
SpherVol := 4 * Pi * R * R;
```

Procedure Statement

A procedure statement serves to activate a previously defined user-defined procedure or a pre-defined standard procedure. The statement consists of a procedure identifier, optionally followed by a parameter list, which is a list of variables or expressions separated by commas and enclosed in parentheses. When the procedure statement is encountered during program execution, control is transferred to the named procedure, and the value (or the address) of possible parameters are transferred to the procedure. When the procedure finishes, program execution continues from the statement following the procedure statement.

Examples:

```
Find(Name,Address);
Sort(Address);
UpperCase(Text);
UpdateCustFile(CustRecord);
```

Goto Statement

A **goto** statement consists of the reserved word **goto** followed by a label identifier. It serves to transfer further processing to that point in the program text which is marked by the label. The following rules should be observed when using **goto** statements:

1) Before use, labels must be declared. The declaration takes place in a label declaration in the declaration part of the block in which the label is used.
2) The scope of a label is the block in which it is declared. It is thus not possible to jump into or out of procedures and functions.

Empty Statement

An 'empty' statement is a statement which consists of no symbols, and which has no effect. It may occur whenever the syntax of Pascal requires a statement but no action is to take place.

Examples:

```
begin end.
while Answer <> '' do;
repeat until KeyPressed; {wait for any key to be hit}
```


Structured Statements

Structured statements are constructs composed of other statements which are to be executed in sequence (compound statements), conditionally (conditional statements), or repeatedly (repetitive statements). The discussion of the **with** statement is deferred to pages 81 pp.

Compound Statement

A compound statement is used if more than one statement is to be executed in a situation where the Pascal syntax allows only one statement to be specified. It consists of any number of statements separated by semi-colons and enclosed within the reserved words **begin** and **end**, and specifies that the component statements are to be executed in the sequence in which they are written.

Example:

```
if Small > Big then
begin
  Tmp := Small;
  Small := Big;
  Big := Tmp;
end;
```

Conditional Statements

A conditional statement selects for execution a single one of its component statements.

If Statement

The **if** statement specifies that a statement be executed only if a certain condition (Boolean expression) is true. If it is false, then either no statement or the statement following the reserved word **else** is to be executed. Notice that **else** must not be preceded by a semicolon.

The syntactic ambiguity arising from the construct:

```
if expr1 then
  if expr2 then
    stmt1
  else
    stmt2
```

is resolved by interpreting the construct as follows:

```
if expr1 then
begin
  if expr2 then
    stmt1
  else
    stmt2
end
```

The **else**-clause part belongs generally to the last **if** statement which has no **else** part.

Examples:

```
if Interest > 25 then
  Usury := True
else
  TakeLoan := OK;

if (Entry < 0) or (Entry > 100) then
begin
  Write('Range is 1 to 100, please re-enter: ');
  Read(Entry);
end;
```

Case Statement

The **case** statement consists of an expression (the selector) and a list of statements, each preceded by a case label of the same type as the selector. It specifies that the one statement be executed whose case label is equal to the current value of the selector. If none of the case labels contain the value of the selector, then either no statement is executed, or, optionally, the statements following the reserved word **else** are executed. The **else** clause is an expansion of standard Pascal.

A case label consists of any number of constants or subranges separated by commas followed by a colon. A subrange is written as two constants separated by the subrange delimiter '..'. The type of the constants must be the same as the type of the selector. The statement following the case label is executed if the value of the selector equals one of the constants or if it lies within one of the subranges.

Valid selector types are all simple types, i.e. all scalar types except real.

Examples:

```
case Operator of
  '+': Result := Answer + Result;
  '-': Result := Answer - Result;
  '*': Result := Answer * Result;
  '/': Result := Answer / Result;
end;

case Year of
  Min..1939: begin
               Time := PreWorldWar2;
               Writeln('The world at peace...');
             end;
  1946..Max: begin
               Time := PostWorldWar2
               Writeln('Building a new world.');
             end;
       else  begin
               Time := WorldWar2;
               Writeln('We are at war');
             end;
end;
```

Repetitive Statements

Repetitive statements specify that certain statements are to be executed repeatedly. If the number of repetitions is known before the repetitions are started, the **for** statement is the appropriate construct to express this situation. Otherwise the **while** or the **repeat** statement should be used.

For Statement

The **for** statement indicates that the component statement is to be repeatedly executed while a progression of values is assigned to a variable which is called the *control variable*. The progression can be ascending: **to** or descending: **downto** the *final value*.

The control variable, the *initial value*, and the *final value* must all be of the same type. Valid types are all simple types, i.e. all scalar types except real.

If the initial value is greater than the final value when using the **to** clause, or if the initial value is less than the final value when using the **downto** clause, the component statement is not executed at all.

Examples:

```
for I := 2 to 100 do if A[I] > Max then Max := A[I];

for I := 1 to NoOfLines do
begin
  Readln(Line);
  if Length(Line) < Limit then
    ShortLines := ShortLines + 1
  else
    LongLines := LongLines + 1
end;
```

Notice that the component statement of a **for** statement must *not* contain assignments to the control variable. If the repetition is to be terminated before the final value is reached, a **goto** statement must be used, although such constructs are not recommended - it is better programming practice use a **while** or a **repeat** statement instead.

Upon completion of a **for** statement, the *control variable* equals the *final value*, unless the loop was not executed at all, in which case no assignment is made to the control variable.

While statement

The expression controlling the repetition must be of type Boolean. The statement is repeatedly executed as long as *expression* is *True*. If its value is false at the beginning, the statement is not executed at all.

Examples:

```
while Size > 1 do Size := Sqrt(Size);

while ThisMonth do
begin
  ThisMonth := CurMonth = SampleMonth;
  Process;
  {process this sample by the Process procedure}
end;
```

Repeat Statement

The expression controlling the repetition must be of type Boolean. The sequence of statements between the reserved words **repeat** and **until** is executed repeatedly until the expression becomes true. As opposed to the **while** statement, the **repeat** statement is always executed at least once, as evaluation af the condition takes place at the end of the loop.

Example:

```
repeat
  Write(^M,'Delete this item? (Y/N)');
  Read(Answer);
until UpCase(Answer) in ['Y','N'];
```

Notes:

Chapter 8
SCALAR AND SUBRANGE TYPES

The basic data types of Pascal are the scalar types. Scalar types constitute a finite and linear ordered set of values. Although the standard type *Real* is included as a scalar type, it does not conform to this definition. Therefore, *Reals* may not always be used in the same context as other scalar types.

Scalar Type

Apart from the standard scalar types (*Integer, Real, Boolean, Char*, and *Byte*), Pascal supports user defined scalar types, also called declared scalar types. The definition of a scalar type specifies, in order, all of its possible values. The values of the new type will be represented by identifiers, which will be the constants of the new type.

Examples:

```
type
  Operator = (Plus,Minus,Multi,Divide);
  Day      = (Mon,Tues,Wed,Thur,Fri,Sat,Sun);
  Month    = (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
  Card     = (Club,Diamond,Heart,Spade);
```

Variables of the above type *Card* can assume one of four values, namely *Club*, *Diamond*, *Heart*, or *Spade*. You are already acquainted with the standard scalar type Boolean which is defined as:

```
type
  Boolean = (False,True);
```

The relational operators =, <>, >, <, >=, and <= can be applied to all scalar types, as long as both operands are of the same type (reals and integers may be mixed). The ordering of the scalar type is used as the basis of the comparison, i.e. the order in which the values are introduced in the type definition. For the above type card, the following is true:

```
Club < Diamond < Heart < Spade
```

The following standard functions can be used with arguments of scalar type:

`Succ(Diamond)`	The successor of *Diamond* (Heart).
`Pred(Diamond)`	The predecessor of *Diamond* (Club).
`Ord(Diamond)`	The ordinal value of *Diamond* (1 [as the ordinal value of the first value of a scalar type is 0]).

The result type of *Succ* and *Pred* is the same as the argument type. The result type of *Ord* is *Integer*.

Subrange Type

A type may be defined as a subrange of another already defined scalar type. Such types are called subranges. The definition of a subrange simply specifies the least and the largest value in the subrange. The first constant specifies the lower bound and must not be greater than the second constant, the upper bound. A subrange of type *Real* is not allowed.

Examples:

```
type
  HemiSphere   = (North, South, East, West);
  World        = (East..West)
  CompassRange = 0..360;
  Upper        = 'A'..'Z';
  Lower        = 'a'..'z';
  Degree       = (Celc, Fahr, Ream, Kelv);
  Wine         = (Red, White, Rose, Sparkling);
```

The type *World* is a subrange of the scalar type *HemiSphere* (called the *associated scalar type*). The associated scalar type of *Compassrange* is *Integer*, and the associated scalar type of *Upper* and *Lower* is *Char*.

You already know the standard subrange type *Byte*, which is defined as:

```
type
  Byte = 0..255;
```

A subrange type retains all the properties of its associated scalar type, being restricted only in its range of values.

The use of defined scalar types and subrange types is strongly recommended as it greatly improves the readability of programs. Furthermore, run time checks may be included in the program code (see page 65) to verify the values assigned to defined scalar variables and subrange variables. Another advantage of defined types and subrange types is that they often save memory. TURBO Pascal allocates only one byte of memory for variables of a defined scalar type or a subrange type with a total number of elements less than 256. Similarly, integer subrange variables, where lower and upper bounds are both within the range 0 through 255, occupy only one byte of memory.

Type Conversion

The *Ord* function may be used to convert scalar types into values of type integer. Standard Pascal does not provide a way to reverse this process, i.e. a way of converting an integer into a scalar value.

In TURBO Pascal, a value of a scalar type may be converted into a value of another scalar type, with the same ordinal value, by means of the *Retype* facility. Retyping is achieved by using the type identifier of the desired type as a function designator followed by one parameter enclosed in parentheses. The parameter may be a value of any scalar type except *Real*. Assuming the type definitions on pages 63 and 64 , then:

```
Integer(Heart) = 2
Month(10)      = Nov
HemiSphere(2)  = East
Upper(14)      = 'O'
Degree(3)      = Kelv
Char(78)       = 'N'
Integer('7')   = 55
```

Range Checking

The generation of code to perform run-time range checks on scalar and subrange variables is controlled with the **R** compiler directive. The default setting is { $R-}, i.e. no checking is performed. When an assignment is made to a scalar or a subrange variable while this directive is active ({ $R + }), assignment values are checked to be within range. It is recommended to use this setting as long as a program is not fully debugged.

Example:

```
program Rangecheck;
type
  Digit = 0..9;
Var
  Dig1,Dig2,Dig3: digit;
begin
  Dig1 := 5;            {valid}
  Dig2 := Dig1 + 3;     {valid as Dig1 + 3 < = 9}
  Dig3 := 47;           {invalid but causes no error}
  {$R+} Dig3 := 55;     {invalid and causes a run time error}
  {$R-} Dig3 := 167;    {invalid but causes no error}
end.
```


Chapter 9
STRING TYPE

TURBO Pascal offers the convenience of **string** types for processing of character strings, i.e. sequences of characters. String types are structured types, and are in many ways similar to **array** types (see chapter 10). There is, however, one major difference between these: the number of characters in a string (i.e. the *length* of the string) may vary dynamically between 0 and a specified upper limit, whereas the number of elements in an array is fixed.

String Type Definition

The definition of a string type must specify the maximum number of characters it can contain, i.e. the maximum length of strings of that type. The definition consists of the reserved word **string** followed by the maximum length enclosed in square brackets. The length is specified by an integer constant in the range 1 through 255. Notice that strings do not have a default length; the length must always be specified.

Example:

```
type
  FileName = string[14];
  ScreenLine = string[80];
```

String variables occupy the defined maximum length in memory plus one byte which contains the current length of the variable. The individual characters within a string are indexed from 1 through the length of the string.

String Expressions

Strings are manipulated by the use of *string expressions*. String expressions consist of string constants, string variables, function designators, and operators.

The plus-sign may be used to concatenate strings. The *Concat* function (see page 71) performs the same function, but the + operator is often more convenient. If the length of the result is greater than 255, a run-time error occurs.

Example:

```
'TURBO ' + 'Pascal'          = 'TURBO Pascal'
'123' + '.' + '456'          = '123.456'
'A ' + 'B' + ' C ' + 'D '    = 'A B C D '
```

The relational operators =, <>, >, <, >=, and <= are lower in precedence than the concatenation operator. When applied to string operands, the result is a Boolean value (*True* or *False*). When comparing two strings, single characters are compared from the left to the right according to their ASCII values. If the strings are of different length, but equal up to and including the last character of the shortest string, then the shortest string is considered the smaller. Strings are equal only if their lengths as well as their contents are identical.

Examples:

```
'A' < 'B'                                  is true
'A' > 'b'                                  is false
'2' < '12'                                 is false
'TURBO' = 'TURBO'                          is true
'TURBO ' = 'TURBO'                         is false
'Pascal Compiler' < 'Pascal compiler'      is true
```

String Assignment

The assignment operator is used to assign the value of a string expression to a string variable.

Example:

```
Age := 'fiftieth';
Line := 'Many happy returns on your ' +  Age + ' birthday.'
```

If the maximum length of a string variable is exceeded (by assigning too many characters to the variable), the exceeding characters are truncated. E.g., if the variable *Age* above was declared to be of type **string**[5], then after the assignment, the variable will only contain the five leftmost characters: 'fifth'.

String Procedures

The following standard string procedures are available in TURBO Pascal:

Delete

Syntax: Delete (*St* , *Pos* , *Num*);

Delete removes a substring containing *Num* characters from *St* starting at position *Pos*. *St* is a string variable and both *Pos* and *Num* are integer expressions. If *Pos* is greater than *Length* (*St*), no characters are removed. If an attempt is made to delete characters beyond the end of the string (i.e. *Pos* + *Num* exceeds the length of the string), only characters within the string are deleted. If *Pos* is outside the range 1..255, a run time error occurs.

If *St* has the value 'ABCDEFG' then:
`Delete(St,2,4)` will give *St* the value 'AFG'.
`Delete(St,2,10)` will give *St* the value 'A'.

Insert

Syntax: Insert (*Obj* , *Target* , *Pos*);

Insert inserts the string *Obj* into the string *Target* at the position *Pos*. *Obj* is a string expression, *Target* is a string variable, and *Pos* is an integer expression. If *Pos* is greater than Length(*Target*), then *obj* is concatenated to *Target*. If the result is longer than the maximum length of *Target*, then excess characters will be truncated and *Target* will only contain the leftmost characters. If *Pos* is outside the range 1..255, a run time error occurs.

If *St* has the value 'ABCDEFG' then: `Insert('XX',St,3)` will give *St* the value 'ABXXCDEFG'

Str

Syntax: Str (*Value* , *St*);

The *Str* procedure converts the numeric value of *Value* into a string and stores the result in *St*. *Value* is a write parameter of type integer or of type real, and *St* is a string variable. Write parameters are expressions with special formatting commands (see page 111).

If *I* has the value 1234 then: `Str(I:5,St)` gives *St* the value `' 1234'`.

If *X* has the value 2.5E4 then: `Str(X:10:0,St)` gives *St* the value `'      2500'`.

8-bit systems only: a function using the *Str* procedure must **never** be called by an expression in a *Write* or *Writeln* statement.

Val

Syntax: Val (*St* , *Var* , *Code*);

Val converts the string expression *St* to an integer or a real value (depending on the type of *Var*) and stores this value in *Var*. *St* must be a string expressing a numeric value according to the rules applying to numeric constants (see page 43). Neither leading nor trailing spaces are allowed. *Var* must be an *Integer* or a *Real* variable and *Code* must be an integer variable. If no errors are detected, the variable *Code* is set to 0. Otherwise *Code* is set to the position of the first character in error, and the value of *Var* is undefined.

If *St* has the value '234' then:
`Val(St,I,Result)` gives *I* the value 234 and *Result* the value 0

If *St* has the value '12x' then:
`Val(St,I,Result)` gives *I* an undefined value and *Result* the value 3

If *St* has the value '2.5E4', and *X* is a *Real* variable, then:
`Val(St,X,Result)` gives *X* the value 2500 and *Result* the value 0

8-bit systems only: a function using the *Var* procedure must **never** be called by an expression in a *Write* or *Writeln* statement.

String Functions

The following standard string functions are available in TURBO Pascal:

Copy

Syntax: Copy (*St* , *Pos* , *Num*);

Copy returns a substring containing *Num* characters from *St* starting at position *Pos*. *St* is a string expression and both *Pos* and *Num* are integer expressions. If *Pos* exceeds the length of the string, the empty string is returned. If an attempt is made to get characters beyond the end of the string (i.e. *Pos* + *Num* exceeds the length of the string), only the characters within the string are returned. If *Pos* is outside the range 1..255, a run time error occurs.

If *St* has the value 'ABCDEFG' then:

`Copy(St,3,2)`	returns the value 'CD'
`Copy(St,4,10)`	returns the value 'DEFG'
`Copy(St,4,2)`	returns the value 'DE'

Concat

Syntax: Concat (*St1* , *St2* { , *StN* });

The *Concat* function returns is a string which is the concatenation of its arguments in the order in which they are specified. The arguments may be any number of string expressions separated by commas (*St1*, *St2* .. *StN*). If the length of the result is greater than 255, a run-time error occurs. As explained in page 68 , the + operator can be used to obtain the same result, often more conveniently. *Concat* is included only to maintain compatibility with other Pascal compilers.

If *St1* has the value 'TURBO' and *St2* the value 'is fastest' then:

```
Concat(Stl,' PASCAL ',St2)
```

returns the value 'TURBO PASCAL is fastest'

Length

Syntax: Length (*St*);

Returns the length of the string expression *St*, i.e. the number of characters in *St*. The type of the result is integer.

If *St* has the value '123456789' then:
`Length(St)` returns the value 9

Pos

Syntax: Pos (*Obj* , *Target*);

The *Pos* function scans the string *Target* to find the first occurrence of *Obj* within *Target*. *Obj* and *Target* are string expressions, and the type of the result is integer. The result is an integer denoting the position within *Target* of the first character of the matched pattern. The position of the first character in a string is 1. If the pattern is not found, *Pos* returns 0.

If *St* has the value 'ABCDEFG' then
`Pos('DE',St)` returns the value 4
`Pos('H',St)` returns the value 0

Strings and Characters

String types and the standard scalar type *Char* are compatible. Thus, whenever a string value is expected, a char value may be specified instead and vice versa. Furthermore, strings and characters may be mixed in expressions. When a character is assigned a string value, the length of the string must be exactly one; otherwise a run-time error occurs.

The characters of a string variable may be accessed individually through string indexing. This is achieved by appending an index expression of type integer, enclosed in square brackets, to the string variable.

Examples:
```
Buffer[5]
Line[Length(Line)-1]
Ord(Line[0])
```

As the first character of the string (at index 0) contains the length of the string, `Length(String)` is the same as `Ord(String[0])`. If assignment is made to the length indicator, it is the responsibility of the programmer to check that it is less than the maximum length of the string variable. When the range check compiler directive **R** is active ({$R+}), code is generated which insures that the value of a string index expression does not exceed the maximum length of the string variable. It is, however, still possible to index a string beyond its current dynamic length. The characters thus read are random, and assignments beyond the current length will not affect the actual value of the string variable.

Notes:

Chapter 10
ARRAY TYPE

An array is a structured type consisting of a fixed number of components which are all of the same type, called the *component type* or the *base type*. Each component can be explicitly accessed by indices into the array. Indices are expressions of any scalar type placed in square brackets suffixed to the *array identifier*, and their type is called the *index type*.

Array Definition

The definition of an array consists of the the reserved word **array** followed by the index type, enclosed in square brackets, followed by the reserved word **of**, followed by the component type.

Examples:

```
type
  Day = (Mon,Tue,Wed,Thu,Fri,Sat,Sun)
Var
  WorkHour : array[1..8] of Integer;
  Week     : array[1..7] of Day;

type
  Players  = (Player1,Player2,Player3,Player4);
  Hand     = (One,Two,Pair,TwoPair,Three,Straight,
              Flush,FullHouse,Four,StraightFlush,RSF);
  LegalBid = 1..200;
  Bid      = array[Players] of LegalBid;
Var
  Player   : array[Players] of Hand;
  Pot      : Bid;
```

An array component is accessed by suffixing an index enclosed in square brackets to the array variable identifier:

```
Player[Player3] := FullHouse;
Pot[Player3] := 100;
Player[Player4] := Flush;
Pot[Player4] := 50;
```

As assignment is allowed between any two variables of identical type, entire arrays can be copied with a single assignment statement.

The R compiler directive controls the generation of code which will perform range checks on array index expressions at run-time. The default mode is passive, i.e. { $R-}, and the { $R + } setting causes all index expressions to be checked against the limits of their index type.

Multidimensional Arrays

The component type of an array may be any data type, i.e. the component type may be another array. Such a structure is called a *multidimensional array*.

Example:

```
type
  Card     = (Two,Three,Four,Five,Six,Seven,Eight,Nine,
              Ten,Knight,Queen,King,Ace);
  Suit     = (Hearts,Spade,Clubs,Diamonds);
  AllCards = array[Suit] of array[1..13] of Card;
Var
  Deck: AllCards;
```

A multi-dimensional array may be defined more conveniently by specifying the multiple indices thus:

```
type
  AllCards = array[Suit,1..13] of Card;
```

A similar abbreviation may be used when selecting an array component:

`Deck[Hearts,10]` is equivalent to `Deck[Hearts][10]`

It is, of course, possible to define multi-dimensional arrays in terms of previously defined array types.

Example:

```
type
  Pupils       = string[20];
  Class        = array[1..30] of Pupils;
  School       = array[1..100] of Class;
Var
  J,P,Vacant   : Integer
  ClassA,
  ClassB       : Class;
  NewTownSchool: School;
```

After these definitions, all of the following assignments are legal:

```
ClassA[J]:='Peter';
NewTownSchool[5][21]:='Peter Brown';
NewTownSchool[8,J]:=NewTownSchool[7,J]; {pupil no. J changed class}
ClassA[Vacant]:=ClassB[P]; {pupil no. P changes Class and number}
```

Character Arrays

Character arrays are arrays with one index and components of the standard scalar type *Char*. Character arrays may be thought of as *strings* with a constant length.

In TURBO Pascal, character arrays may participate in *string* expressions, in which case the array is converted into a string of the length of the array. Thus, arrays may be compared and manipulated in the same way as strings, and string constants may be assigned to character arrays, as long as they are of the same length. String variables and values computed from string expressions cannot be assigned to character arrays.

Predefined Arrays

TURBO Pascal offers two predefined arrays of type *Byte*, called *Mem* and *Port*, which are used to access CPU memory and data ports. These are discussed in chapters 20, 21, and 22.

Notes:

Chapter 11
RECORD TYPE

A **record** is a structure consisting of a fixed number of components, called *fields*. Fields may be of different type and each field is given a name, the *field identifier*, which is used to select it.

Record Definition

The definition of a record type consists of the reserved word **record** succeeded by a *field list* and terminated by the reserved word **end**. The field list is a sequence of *record sections* separated by semi-colons, each consisting of one or more identifiers separated by commas, followed by a colon and either a type *identifier* or a type *descriptor*. Each record section thus specifies the identifier and type of one or more fields.

Example:

```
type
  DaysOfMonth = 1..31;
  Date =        record
                  Day:   DaysOfMonth;
                  Month: (Jan,Feb,Mar,Apr,May,Jun,
                          July,Aug,Sep,Oct,Nov,Dec);
                  Year:  1900..1999;
                end;
Var
  Birth: Date;
  WorkDay: array[1..5] of date;
```

Day, *Month*, and *Year*are field identifiers. A field identifier must be unique only within the record in which it is defined. A field is referenced by the variable identifier and the field identifier separated by a period.

Examples:

```
Birth.Month := Jun;
Birth.Year := 1950;
WorkDay[Current] := WorkDay[Current-1];
```

Note that, similar to array types, assignment is allowed between entire records of identical types. As record components may be of any type, constructs like the following record of records of records are possible:

```
type
  Name   = record
             FamilyName: string[32];
             ChristianNames: array[1..3] of string[16];
           end;
  Rate   = record
             NormalRate, OverTime,
             NightTime, Weekend: Integer
           end;
  Date   = record
             Day:   1..31;
             Month: (Jan,Feb,Mar,Apr,May,Jun,
                     July,Aug,Sep,Oct,Nov,Dec);
             Year:  1900..1999;
           end;
  Person = record
             ID: Name;
             Time: Date;
           end;
  Wages  = record
             Individual: Person;
             Cost: Rate;
           end

  Var  Salary, Fee: Wages;
```

Assuming these definitions, the following assignments are legal:

```
Salary := Fee;
Salary.Cost.Overtime := 950;
Salary.Individual.Time := Fee.Individual.Time;
Salary.Individual.ID.FamilyName := 'Smith'
```


With Statement

The use of records as describes above does sometimes result in rather lengthy statements; it would often be easier if we could access individual fields in a record as if they were simple variables. This is the function of the **with** statement: it 'opens up' a record so that field identifiers may be used as variable identifiers.

A **with** statement consists of the reserved word **with** followed by a list of record variables separated by commas followed by the reserved word **do** and finally a statement.

Within a **with** statement, a field is designated only by its field identifier, i.e. without the record variable identifier:

```
with Salary do
begin
  Individual := NewEmployee;
  Cost := StandardRates;
end;
```

Records may be *nested* within **with** statements, i.e. records of records may be 'opened' as shown here:

```
with Salary, Individual, ID do
begin
  FamilyName := 'Smith';
  ChristianNames[1] := 'James';
end
```

This is equivalent to:

```
with Salary do with Individual do with ID do
...
```

The maximum 'depth' of this nesting of **with** sentences, i.e. the maximum number of records which may be 'opened' within one block, depends on your implementation and is discussed in chapters 20, 21, and 22.

Variant Records

The syntax of a record type also provides for a variant part, i.e. alternative record structures which allows fields of a record to consist of a different number and different types of components, usually depending on the value of a *tag field*.

A variant part consists of a *tag-field* of a previously defined type, whose values determine the variant, followed by labels corresponding to each possible value of the tag field. Each label heads a *field list* which defines the type of the variant corresponding to the label.

Assuming the existence of the type:

```
Origin = (Citizen, Alien);
```

and of the types *Name* and *Date*, the following record allows the field *CitizenShip* to have different structures depending on whether the value of the field is *Citizen* or *Alien*:

```
type
  Person = record
             PersonName: Name;
             BirthDate: Date;
             case CitizenShip: Origin of
               Citizen: (BirthPlace: Name);
               Alien:   (CountryOfOrigin: Name;
                         DateOfEntry: Date;
                         PermittedUntil: Date;
                         PortOfEntry: Name);
           end;
```

In this variant record definition, the tag-field is an explicit field which may be selected and updated like any other field. Thus, if *Passenger* is a variable of type *Person*, statements like the following are perfectly legal:

```
Passenger.CitizenShip := Citizen;

with Passenger, PersonName do
  if CitizenShip = Alien then writeln(FamilyName);
```


The fixed part of a record, i.e. the part containing the common fields, must always precede the *variant part*. In the above example, the fields *PersonName* and *BirthDate* are the fixed fields. A record can only have one variant part. In a variant, the parentheses must be present, even if they will enclose nothing.

The maintenance of tag field values is the responsibility of the programmer and not of TURBO Pascal. Thus, in the *Person* type above, the field *DateOfEntry* can be accessed even if the value of the tag field *CitizenShip* is not *Alien*. Actually, the tag field identifier may be omitted altogether, leaving only the type identifier. Such record variants are known as *free unions*, as opposed to record variants with tag fields which are called *discriminated unions*. The use of free unions is infrequent and should only be practiced by experienced programmers.

Notes:

Chapter 12
SET TYPE

A **set** is a collection of related objects which may be thought of as a whole. Each object in such a set is called a *member* or an *element* of the set. Examples of sets could be:

1) All integers between 0 and 100
2) The letters of the alphabet
3) The consonants of the alphabet

Two sets are equal if and only if their elements are the same. There is no ordering involved, so the sets [1,3,5], [5,3,1] and [3,5,1] are all equal. If the members of one set are also members of another set, then the first set is said to be included in the second. In the examples above, 3) is included in 2).

There are three operations involving sets, similar to the operations addition, multiplication and subtraction operations on numbers:

The *union* (or sum) of two sets A and B (written A + B) is the set whose members are members of either A or B. For instance, the union of [1,3,5,7] and [2,3,4] is [1,2,3,4,5,7].

The *intersection* (or product) of two sets A and B (written A*B) is the set whose members are the members of both A and B. Thus, the intersection of [1,3,4,5,7] and [2,3,4] is [3,4].

The *relative complement* of B with respect to A (written A-B) is the set whose members are members of A but not of B. For instance, [1,3,5,7]-[2,3,4] is [1,5,7].

Set Type Definition

Although in mathematics there are no restrictions on the objects which may be members of a set, Pascal only offers a restricted form of sets. The members of a set must all be of the same type, called the *base type*, and the base type must be a simple type, i.e. any scalar type except real. A set type is introduced by the reserved words **set of** followed by a simple type.

Examples:

```
type
  DaysOfMonth = set of 0..31;
  WorkWeek = set of Mon..Fri;
  Letter = set of 'A'..'Z';
  AdditiveColors = set of (Red,Green,Blue);
  Characters = set of Char;
```

In TURBO Pascal, the maximum number of elements in a set is 256, and the ordinal values of the base type must be within the range 0 through 255.

Set Expressions

Set values may be computed from other set values through set expressions. Set expressions consist of set constants, set variables, set constructors, and set operators.

Set Constructors

A set constructor consists of one or more element specifications, separated by commas, and enclosed in square brackets. An element specification is an expression of the same type as the base type of the set, or a range expressed as two such expressions separated by two consecutive periods (..).

Examples:

```
['T','U','R','B','O']
[X,Y]
[X..Y]
[1..5]
['A'..'Z','a'..'z','0'..'9']
[1,3..10,12]
[]
```

The last example shows *the empty set*, which, as it contains no expressions to indicate its base type, is compatible with all set types. The set [1..5] is equivalent to the set [1,2,3,4,5]. If X > Y then [X..Y] denotes the empty set.

Set Operators

The rules of composition specify set operator precedency according to the following three classes of operators:

1) `*` Set intersection.

2) `+` Set union.
   `-` Set difference.

3) `=` Test on equality.
   `<>` Test on inequality.
   `>=` *True* if all members of the second operand are included in the first operand.
   `<=` *True* if all members of the first operand are included in the second operand.
   `IN` Test on set membership. The second operand is of a set type, and the first operand is an expression of the same type as the base type of the set. The result is true if the first operand is a member of the second operand, otherwise it is false.

Set disjunction (when two sets contain no common members) may be expressed as:

```
A * B = [];
```

that is, the intersection between the two sets is the empty set. Set expressions are often useful to clarify complicated tests. For instance, the test:

```
if (Ch='T') or (Ch='U') or (Ch='R') or (Ch='B') or (Ch='O')
```

can be expressed much clearer as:

```
Ch in ['T','U','R','B','O']
```

And the test:

```
if (Ch >= '0') and (Ch <= '9') then ...
```

is better expressed as:

```
if Ch in['0'..'9'] then ...
```

Set Assignments

Values resulting from set expressions are assigned to set variables using the assignment operator `:=`.

Examples:

```
type
  ASCII = set of 0..127;
Var
  NoPrint,Print,AllChars: ASCII;
begin
  AllChars := [0..127];
  NoPrint := [0..31,127];
  Print := AllChars - NoPrint;
end.
```


Chapter 13
TYPED CONSTANTS

Typed constants are a TURBO specialty. A typed constant may be used exactly like a variable of the same type. Typed constants may thus be used as 'initialized variables', because the value of a typed constant is defined, whereas the value of a variable is undefined until an assignment is made. Care should be taken, of course, not to assign values to typed constants whose values are actually meant to be **constant**.

The use of a typed constant saves code if the constant is used often in a program, because a typed constant is included in the program code only once, whereas an untyped constant is included every time it is used.

Typed constants are defined like untyped constants (see page 48), except that the definition specifies not only the *value* of the constant but also the *type*. In the definition the typed constant identifier is succeeded by a colon and a type identifier, which is then followed by an equal sign and the actual constant.

Unstructured Typed Constants

An unstructured typed constant is a constant defined as one of the scalar types:

```
const
  NumberOfCars: Integer = 1267;
  Interest: Real = 12.67;
  Heading: string[7] = 'SECTION';
  Xon: Char = ^Q;
```

Contrary to untyped constants, a typed constant may be used in place of a variable as a variable parameter to a procedure or a function. As a typed constant is actually a variable with a constant value, it cannot be used in the definition of other constants or types. Thus, as *Min* and *Max* are typed constants, the following construct is **illegal**:

```
const
  Min: Integer = 0;
  Max: Integer = 50;
type
  Range: array[Min..Max] of integer
```

Structured Typed Constants

Structured constants comprise *array constants, record constants,* and *set constants.* They are often used to provide initialized tables and sets for tests, conversions, mapping functions, etc. The following sections describe each type in detail.

Array Constants

The definition of an array constant consists of the constant identifier succeeded by a colon and the type identifier of a previously defined array type followed by an equal sign and the constant value expressed as a set of constants separated by commas and enclosed in parentheses.

Examples:

```
type
  Status    = (Active,Passive,Waiting);
  StringRep = array[Status] of string[7];
const
  Stat: StringRep = ('active','passive','waiting');
```

The example defines the array constants *Stat*, which may be used to convert values of the scalar type *Status* into their corresponding string representations. The components of *Stat* are:

```
Stat[Active]  = 'active'
Stat[Passive] = 'passive'
Stat[Waiting] = 'waiting'
```

The component type of an array constant may be any type except *File* types and *Pointer* types. Character array constants may be specified both as single characters and as strings. Thus, the definition:


```
const
  Digits: array[0..9] of Char =
  ('0','1','2','3','4','5','6','7','8','9');
```

may be expressed more conveniently as:

```
const
  Digits: array[0..9] of Char = '0123456789';
```

Multi-dimensional Array Constants

Multi-dimensional array constants are defined by enclosing the constants of each dimension in separate sets of parentheses, separated by commas. The innermost constants correspond to the rightmost dimensions.

Example:

```
type
  Cube = array[0..1,0..1,0..1] of integer;
const
  Maze: Cube = (((0,1),(2,3)),((4,5),(6,7)));
begin
  Writeln(Maze[0,0,0],' = 0');
  Writeln(Maze[0,0,1],' = 1');
  Writeln(Maze[0,1,0],' = 2');
  Writeln(Maze[0,1,1],' = 3');
  Writeln(Maze[1,0,0],' = 4');
  Writeln(Maze[1,0,1],' = 5');
  Writeln(Maze[1,1,0],' = 6');
  Writeln(Maze[1,1,1],' = 7');
end.
```

Record Constants

The definition of a record constant consists of the constant identifier succeeded by a colon and the type identifier of a previously defined record type followed by an equal sign and the constant value expressed as a list of field constants separated by semi-colons and enclosed in parentheses.

Examples:

```
type
  Point      = record
                 X,Y,Z: integer;
               end;
  OS         = (CPM80,CPM86,MSDOS,Unix);
  UI         = (CCP,SomethingElse,MenuMaster);
  Computer   = record
                 OperatingSystems: array[1..4] of OS;
                 UserInterface: UI;
               end;
const
  Origo: Point   = (X:0; Y:0; Z:0);
  SuperComp: Computer =
             (OperatingSystems: (CPM80,CPM86,MSDOS,Unix);
             UserInterface: MenuMaster);
  Planel: array[1..3] of Point =
      ((X:1;Y:4;Z:5),(X:10;Y:-78;Z:45),(X:100;Y:10;Z:-7));
```

The field constants must be specified in the same order as they appear in the definition of the record type. If a record contains fields of file types or pointer types, then constants of that record type cannot be specified. If a record constant contains a variant, then it is the responsibility of the programmer to specify only the fields of the valid variant. If the variant contains a tag field, then its value must be specified.

Set Constants

A set constant consists of one or more element specifications separated by commas, and enclosed in square brackets. An element specification must be a constant or a range expression consisting of two constants separated by two consecutive periods (..).

Example:

```
type
  Up  = set of 'A'..'Z';
  Low = set of 'a'..'z';
const
  UpperCase: Up   = ['A'..'Z'];
  Vocals   : Low  = ['a','e','i','o','u','y'];
  Delimiter: set of Char =
                   [' '..'/',':'..'?','['..'`','{'..' '];
```


Chapter 14
FILE TYPES

Files provide a program with channels through which it can pass data. A file can either be a *disk file*, in which case data is written to and read from a magnetic device of some type, or a *logical device*, such as the pre-defined files *Input* and *Output* which refer to the computer's standard I/O channels; the keyboard and the screen.

A *file* consists of a sequence of components of equal type. The number of components in a file (the *size* of the file) is not determined by the definition of the file; instead the Pascal system keeps track of file accesses through a *file pointer*, and each time a component is written to or read from a file, the file pointer of that file is advanced to the next component. As all components of a file are of equal length, the position of a specific component can be calculated. Thus, the file pointer can be moved to any component in the file, providing random access to any element of the file.

File Type Definition

A file type is defined by the reserved words **file of** followed by the type of the components of the file, and a file identifier is declared by the same words followed by the identifier of a previously defined file type.

Examples:

```
type
  ProductName = string[80];
  Product = file of record
                      Name: ProductName;
                      ItemNumber: Real;
                      InStock: Real;
                      MinStock: Real;
                      Supplier: Integer;
                    end;
Var
  ProductFile: Product;
  ProductNames: file of ProductName;
```