The component type of a file may be any type, except a file type. (that is, with reference to the example above, **file of** *Product* is **not** allowed). File variables may appear in neither assignments nor expressions.

Operations on Files

The following sections describe the procedures available for file handling. The identifier *FilVar* used throughout denotes a file variable identifier declared as described above.

Assign

Syntax: Assign(*FilVar, Str*);

Str is a string expression yielding any legal file name. This file name is assigned to the file variable *FilVar*, and all further operation on *FilVar* will operate on the disk file *Str*. Assign should never be used on a file which is in use.

Rewrite

Syntax: Rewrite(*FilVar*);

A new disk file of the name assigned to the file variable *FilVar* is created and prepared for processing, and the file pointer is set to the beginning of the file, i.e. component no. 0. Any previously existing file with the same name is erased. A disk file created by rewrite is initially empty, i.e. it contains no elements.

Reset

Syntax: Reset(*FilVar*);

The disk file of the name assigned to the file variable *FilVar* is prepared for processing, and the file pointer is set to the beginning of the file, i.e. component no. 0. *FilVar* must name an existing file, otherwise an I/O error occurs.

Read

Syntax: Read(*FilVar, Var*);

Var denotes one or more variables of the component type of *FilVar*, separated by commas. Each variable is read from the disk file, and following each read operation, the file pointer is advanced to the next component.

Write

Syntax: Write(*FilVar, Var*);

Var denotes one or more variables of the component type of *FilVar*, separated by commas. Each variable is written to the disk file, and following each write operation, the file pointer is advanced to the next component.

Seek

Syntax: Seek(*FilVar, n*);

Seek moves the file pointer is moved to the n'th component of the file denoted by *FilVar*. *n* is an integer expression. The position of the first component is 0. Note that in order to expand a file it is possible to *seek* one component beyond the last component. The statement

```
Seek(FilVar, FileSize(FilVar));
```

thus places the file pointer at the end of the file (*FileSize* returns the number of components in the file, and as the components are numbered from zero, the returned number is one greater than the number of the last component).

ClibPDF - www.fastio.com

Flush

Syntax: Flush(*FilVar*);

Flush empties the internal sector buffer of the disk file *FilVar*, and thus assures that the sector buffer is written to the disk if any write operations have taken place since the last disk update. *Flush* also insures that the next read operation will actually perform a physical read from the disk file. *Flush* should never be used on a closed file.

Close

Syntax: Close(*FilVar*);

The disk file associated with *FilVar* is closed, and the disk directory is updated to reflect the new status of the file. Notice that it is necessary to *Close* a file, even if it has only been read from—you would otherwise quickly run out of file handles.

Erase

Syntax: Erase(*FilVar*);

The disk file associated with *FilVar* is erased. If the file is open, i.e. if the file has been reset or rewritten but not closed, it is good programming practice to *close* the file before erasing it.

Rename

Syntax: Rename(*FilVar, Str*);

The disk file associated with *FilVar* is renamed to a new name given by the string expression *Str*. The disk directory is updated to show the new name of the file, and further operations on *FilVar* will operate on the file with the new name. *Rename* should never be used on an open file.

Notice that it is the programmer's responsibility to assure that the file named by *Str* does not already exist. If it does, multiple occurrences of the same name may result. The following function returns *True* if the file name passed as a parameter exists, otherwise it returns *False*:

```
type
  Name=string[66];
:
:
function Exist(FileName: Name): boolean;
Var
  Fil: file;
begin
  Assign(Fil, FileName);
  {$I-}
  Reset(Fil);
  {$I+}
  Exist := (IOresult = 0)
end;
```

File Standard Functions

The following standard functions are applicable to files:

EOF

Syntax: EOF(*FilVar*);

A Boolean function which returns *True* if the file pointer is positioned at the end of the disk file, i.e. beyond the last component of the file. If not, *EOF* returns *False*.

FilePos

Syntax: FilePos(*FilVar*);

An integer function which returns the current position of the file pointer. The first component of a file is 0.

ClibPDF - www.fastio.com

FileSize

Syntax: FileSize(*FilVar*);

An integer function which returns the size of the disk file expressed as the number of components in the file. If *FileSize(FilVar)* is zero, the file is empty.

Using Files

Before using a file, the *Assign* procedure must be called to assign the file name to a file variable. Before input and/or output operations are performed, the file must be opened with a call to *Rewrite* or *Reset*. This call will set the file pointer to point to the first component of the disk file, i.e. *FilePos(FilVar)* = 0. After *Rewrite*, *FileSize(FilVar)* is 0.

A disk file can be expanded only by adding components to the end of the existing file. The file pointer can be moved to the end of the file by executing the following sentence:

```
Seek(FilVar, FileSize(FilVar));
```

When a program has finished its input/output operations on a file, it should always call the *Close* procedure. Failure to do so may result in loss of data, as the disk directory is not properly updated.

The program below creates a disk file called *PRODUCTS.DTA*, and writes 100 records of the type *Product* to the file. This initializes the file for subsequent random access (i.e. records may be read and written anywhere in the file).


```
program InitProductFile;
const
  MaxNumberOfProducts = 100;
type
  ProductName = string[20];
  Product = record
              Name: ProductName;
              ItemNumber: Integer;
              InStock: Real;
              Supplier: Integer;
            end;
Var
  ProductFile: file of Product;
  ProductRec: Product;
  I: Integer;
begin
  Assign(ProductFile,'PRODUCT.DTA');
  Rewrite(ProductFile); {open the file and delete any data}
  with ProductRec do
  begin
    Name := ''; InStock := 0; Supplier := 0;
    for I := 1 to MaxNumberOfProducts do
    begin
      ItemNumber := I;
      Write(ProductFile,ProductRec);
    end;
  end;
  Close(ProductFile);
end.
```

The following program demonstrates the use of *Sseek* on random files. The program is used to update the *ProductFile* created by the program in the previous example.

ClibPDF - www.fastio.com

```
program UpDateProductFile;
const
  MaxNumberOfProducts = 100;
type
  ProductName = string[20];
  Product = record
              Name: ProductName;
              ItemNumber: Integer;
              InStock: Real;
              Supplier: Integer;
            end;
Var
  ProductFile: file of Product;
  ProductRec: Product;
  I, Pnr: Integer;
begin
  Assign(ProductFile,'PRODUCT.DTA'); Reset(ProductFile);
  ClrScr;
  Write('Enter product number (0= stop) '); Readln(Pnr);
  while Pnr in [1..MaxNumberOfProducts] do
  begin
    Seek(ProductFile,Pnr-1); Read(ProductFile,ProductRec);
    with ProductRec do
    begin
      Write('Enter name of product (',Name:20,')  ');
      Readln(Name);
      Write('Enter number in stock (',InStock:20:0,')  ');
      Readln(InStock);
      Write('Enter supplier number (',Supplier:20,')  ');
      Readln(Supplier);
      ItemNumber:=Pnr;
    end;
    Seek(ProductFile,Pnr-1);
    Write(ProductFile,ProductRec);
    ClrScr; Writeln;
    Write('Enter product number (0= stop) '); Readln(Pnr);
  end;
  Close(ProductFile);
end.
```


Text Files

Unlike all other file types, *text files* are not simply sequences of values of some type. Although the basic components of a text file are characters, they are structured into lines, each line being terminated by an *end-of-line* marker (a CR/LF sequence). The file is further ended by an *end-of-file* marker (a Ctrl-Z). As the length of lines may vary, the position of a given line in a file cannot be calculated. Text files can therefore only be processed sequentially. Furthermore, input and output cannot be performed simultaneously to a text file.

Operations on Text Files

A text file variable is declared by referring to the standard type identifier *Text*. Subsequent file operations must be preceded by a call to *Assign* and a call to *Reset* or *Rewrite* must furthermore precede input or output operations.

Rewrite is used to create a new text file, and the only operation then allowed on the file is the appending of new components to the end of the file. *Reset* is used to open an existing file for reading, and the only operation allowed on the file is sequential reading. When a new textfile is closed, an end-of-file mark is automatically appended to the file.

Character input and output on text files is made with the standard procedures *Read* and *Write*. Lines are processed with the special text file operators *Readln*, *Writeln*, and *Eoln*:

ReadLn

Syntax: Readln(*Filvar*);

Skips to the beginning of the next line, i.e. skips all characters up to and including the next CR/LF sequence.

WriteLn

Syntax: WriteLn(*Filvar*);

Writes a line marker, i.e. a CR/LF sequence, to the textfile.

ClibPDF - www.fastio.com

Eoln

Syntax: Eoln(*Filvar*);

A Boolean function which returns *True* if the end of the current line has been reached, i.e. if the file pointer is positioned at the CR character of the CR/LF line marker. If *EOF(Filvar)* is true, *Eoln(Filvar)* is also true.

SeekEoln

Syntax: SeekEoln(*FilVar*);

Similar to *Eoln*, except that it skips blanks and TABs before it tests for an end-of-line marker. The type of the result is boolean.

SeekEof

Syntax: SeekEof(*FilVar*);

Similar to *EOF*, except that it skips blanks, TABs, and end-of-line markers (CR/LF sequences) before it tests for an end-of-file marker. The type of the result is boolean.

When applied to a text file, the *EOF* function returns the value *True*

if the file pointer is positioned at the end-of-file mark (the CTRL/Z character ending the file). The *Seek* and *Flush* procedures and the *FilePos* and *FileSize* functions are not applicable to text files.

The following sample program reads a text file from disk and prints it on the pre-defined device *Lst* which is the printer. Words surrounded by Ctrl-S in the file are printed underlined:


```
program TextFileDemo;
Var
  FilVar:       Text;
  Line,
  ExtraLine:    string[255];
  I:            Integer;
  UnderLine:    Boolean;
  FileName:     string[14];
begin
  UnderLine := False;
  Write('Enter name of file to list: ');
  Readln(FileName);
  Assign(FilVar,FileName);
  Reset(FilVar);
  while not Eof(FilVar) do
  begin
    Readln(FilVar,Line);
    I := 1; ExtraLine := '';
    for I := 1 to Length(Line) do
    begin
      if Line[I]<>^S then
      begin
        Write(Lst,Line[I]);
        if UnderLine then ExtraLine := ExtraLine+'_'
        else ExtraLine := ExtraLine+' ';
      end
      else UnderLine := not UnderLine;
    end;
    Write(Lst,^M); Writeln(Lst,ExtraLine);
  end; {while not Eof}
end.
```

Further extensions of the procedures *Read* and *Write*, which facilitate convenient handling of formatted input and output, are described on page 108.

ClibPDF - www.fastio.com

Logical Devices

In TURBO Pascal, external devices such as terminals, printers, and modems are regarded as *logical devices* which are treated like text files. The following logical devices are available:

CON: The console device. Output is sent to the operating system's console output device, usually the CRT, and input is obtained from the console input device, usually the keyboard. Contrary to the TRM: device (see below), the CON: device provides buffered input. In short, this means that each *Read* or *Readln* from a textfile assigned to the CON: device will input an entire line into a line buffer, and that the operator is provided with a set of editing facilities during line input. For more details on console input, please refer to pages 105 and 108 .

TRM: The terminal device. Output is sent to the operating system's console output device, usually the CRT, and input is obtained from the console input device, usually the keyboard. Input characters are echoed, unless they are control characters. The only control character echoed is a carriage return (CR), which is echoed as CR/LF.

KBD: The keyboard device (input only). Input is obtained from the operating system's console input device, usually the keyboard. Input is not echoed.

LST: The list device (output only). Output is sent to the operating system's list device, typically the line printer.

AUX: The auxiliary device. In PC/MS-DOS, this is COM1:; in CP/M it is RDR: and PUN:.

USR: The user device. Output is sent to the user output routine, and input is obtained from the user input routine. For further details on user input and output, please refer to pages 209 , 241 , and 272 .

These logical devices may be accessed through the pre-assigned files discussed on page 105 or they may be assigned to file variables, exactly like a disk file. There is no difference between *Rewrite* and *Reset* on a file assigned to a logical device, *Close* performs no function, and an attempt to *Erase* such a file will cause an I/O error.

The standard functions *Eof* and *Eoln* operate differently on logical devices than on disk files. On a disk file, *Eof* returns *True* when the next character in the file is a Ctrl-Z, or when physical EOF is encountered, and *Eoln* returns *True* when the next character is a CR or a Ctrl-Z. Thus, *Eof* and *Eoln* are in fact 'look ahead' routines.

As you cannot look ahead on a logical device, *Eoln* and *Eof* operate on the *last* character read instead of on the *next* character. In effect, *Eof* returns *True* when the last character read was a Ctrl-Z, and *Eoln* returns *True* when the last character read was a CR or a Ctrl-Z. The following table provides an overview of the operation of *Eoln* and *Eof*:

	On Files	On Logical Devices
Eoln is true if is	**next** character CR or Ctrl-Z or if EOF is true	if **current** character is CR or Ctrl-Z
Eof is true if	**next** character is Ctrl-Z or if physical EOF is met	if **current** character is Ctrl-Z

Table 14-1: Operation of EOLN and Eof

Similarly, the *Readln* procedure works differently on logical devices than on disk files. On a disk file, *Readln* reads all characters up to and including the CR/LF sequence, whereas on a logical device it only reads up to and including the first CR. The reason for this is again the inability to 'look ahead' on logical devices, which means that the system has no way of knowing what character will follow the CR.

Standard Files

As an alternative to assigning text files to logical devices as described above, TURBO Pascal offers a number of pre-declared text files which have already been assigned to specific logical devices and prepared for processing. Thus, the programmer is saved the reset/rewrite and close processes, and the use of these standard files further saves code:

ClibPDF - www.fastio.com

Input
The primary input file. This file is assigned to either the CON: device or to the TRM: device (see below for further detail).

Output
The primary output file. This file is assigned to either the CON: device or to the TRM: device (see below for further detail).

Con Assigned to the console device (CON:).
Trm Assigned to the terminal device (TRM:).
Kbd Assigned to the keyboard device (KBD:).
Lst Assigned to the list device (LST:).
Aux Assigned to the auxiliary device (AUX:).
Usr Assigned to the user device (USR:).

Notice that the use of *Assign, Reset, Rewrite*, and *Close* on these files is illegal.

When the *Read* procedure is used without specifying a file identifier, it always inputs a line, even if some characters still remain to be read from the line buffer, and it ignores Ctrl-Z, forcing the operator to terminate the line with RETURN. The terminating RETURN is not echoed, and internally the line is stored with a Ctrl-Z appended to the end of it. Thus, when less values are specified on the input line than there are parameters in the parameter list, any *Char* variables in excess will be set to Ctrl-Z, strings will be empty, and numeric variables will remain unaltered.

The **B** compiler directive is used to control this 'forced read' feature above. The default state is {$B+}, and in this state, read statements without a file variable will always cause a line to be input from the console. If a {$B-} compiler directive is placed at the beginning of the program (**before** the declaration part), the shortened version of read will act as if the input standard file had been specified, i.e.:

Read(v1,v2,...,vn) equals *Read(input,v1,v2,...,vn)*

and in this case, lines are only input when the line buffer has been emptied. The {$B-} state follows the definition of Standard Pascal I/O, whereas the default {$B+} state, not confirming to the standard in all aspects, provides better control of input operations.

If you don't want input echoed to the screen, you should read from the standard file *Kbd*:

```
Read(Kbd, Var)
```


As the standard files *Input* and *Output* are used very frequently, they are chosen by default if no file identifier is stated. The following list shows the abbreviated text file operations and their equivalents:

Write(Ch)	*Write(Output,Ch)*
Read(Ch)	*Read(Input,Ch)*
Writeln	*Writeln(Output)*
Readln	*Readln(Input)*
Eof	*Eof(Input)*
Eoln	*Eoln(Input)*

The following program shows the use of the standard file *Lst* to list the file *ProductFile* (see page 99) on the printer:

```
program ListProductFile;
const
  MaxNumberOfProducts = 100;
type
  ProductName = string[20];
  Product = record
              Name: ProductName; ItemNumber: Integer;
              InStock: Real;
              Supplier: Integer;
            end;
Var
  ProductFile: file of Product;
  ProductRec: Product; I: Integer;
begin
  Assign(ProductFile,'PRODUCT.DTA'); Reset(ProductFile);
  for I := 1 to MaxNumberOfProducts do
  begin
    Read(ProductFile,ProductRec);
    with ProductRec do
    begin
      if Name<>'' then
        Writeln(Lst,'Item: ',ItemNumber:5,' ', Name:20,
                    '  From: ', Supplier:5,
                    ' Now in stock: ',InStock:0:0);
    end;
  end;
  Close(ProductFile);
end.
```


ClibPDF - www.fastio.com

Text Input and Output

Input and output of data in readable form is done through *text files* as described on page 101. A text file may be assigned to any device, i.e. a disk file or one of the standard I/O devices. Input and output on text files is done with the standard procedures *Read, Readln, Write,* and *Writeln* which use a special syntax for their parameter lists to facilitate maximum flexibility of input and output.

In particular, parameters may be of different types, in which case the I/O procedures provide automatic data conversion to and from the basic *Char* type of text files.

If the first parameter of an I/O procedure is a variable identifier representing a text file, then I/O will act on that file. If not, I/O will act on the standard files *Input* and *Output*. See page 105 for more detail.

Read Procedure

The *Read* procedure provides input of characters, strings, and numeric data. The syntax of the *Read* statement is:

```
Read(Var1,Var2,...,VarN)
```

or

```
Read(FilVar,Var1,Var2,...,VarN)
```

where *Var1, Var2,...,VarN* are variables of type *Char, String, Integer* or *Real*. In the first case, the variables are input from the standard file *Input*, usually the keyboard. In the second case, the variables are input from the text file which is previously assigned to *FilVar* and prepared for reading.

With a variable of type *Char*, *Read* reads one character from the file and assigns that character to the variable. If the file is a disk file, *Eoln* is true if the next character is a CR or a Ctrl-Z, and *Eof* is true if the next character is a Ctrl-Z, or physical end-of-file is met. If the file is a logical device (including the standard files *Input* and *Output*), *Eoln* is true if the character read was a CR or if *Eof* is *True*, and *Eof* is true if the character read was a Ctrl-Z.

With a variable of type **string**, *Read* reads as many characters as allowed by the defined maximum length of the string, unless *Eoln* or *Eof* is reached first. *Eoln* is true if the character read was a CR or if *Eof* is *True*, and *Eof* is true if the last character read is a Ctrl-Z, or physical end-of-file is met.

With a numeric variable (*Integer* or *Real*), *Read* expects a string of characters which complies with the format of a numeric constant of the relevant type as defined on page 43 . Any blanks, TABs, CRs, or LFs preceding the string are skipped. The string must be no longer than 30 characters, and it must be followed by a blank, a TAB, a CR, or a Ctrl-Z. If the string does not conform to the expected format, an I/O error occurs. Otherwise the numeric string is converted to a value of the appropriate type and assigned to the variable. When reading from a disk file, and the input string is ended with a blank or a TAB, the next *Read* or *Readln* will start with the character immediately following that blank or TAB. For both disk files and logical devices, *Eoln* is true if the string was ended with a CR or a Ctrl-Z, and *Eof* is true if the string was ended with a Ctrl-Z.

A special case of numeric input is when *Eoln* or *Eof* is true at the beginning of the *Read* (e.g. if input from the keyboard is only a CR). In that case no new value is assigned to the variable, and the variable retains its former value.

If the input file is assigned to the console device (CON:), or if the standard file *Input* is used in the { $B + } mode (default), special rules apply to the reading of variables. On a call to *Read* or *Readln*, a line is input from the console and stored into a buffer, and the reading of variables then uses this buffer as the input source. This allows for editing during entry. The following editing facilities are available:

BACKSPACE and **DEL**
: Backspaces one character position and deletes the character there. BACKSPACE is usually generated by pressing the key marked BS or BACKSPACE or by pressing Ctrl-H. DEL is usually generated by the key thus marked, or in some cases RUB or RUBOUT.

Esc and **Ctrl-X**
: Backspaces to the beginning of the line and erases all characters input.

ClibPDF - www.fastio.com

Ctrl-D
Recalls one character from the last input line.

Ctrl-R
Recalls the last input line.

RETURN and **Ctrl-M**
Terminates the input line and stores an end-of-line marker (a CR/LF sequence) in the line buffer. This code is generated by pressing the key marked RETURN or ENTER. The CR/LF is **not** echoed to the screen.

Ctrl-Z
Terminates the input line and stores an end-of-file marker (a Ctrl-Z character) in the line buffer.

The input line is stored internally with a Ctrl-Z appended to the end of it. Thus, if fewer values are specified on the input line than the number of variables in *Read*s parameter list, any *Char* variables in excess will be set to Ctrl-Z, *Strings* will be empty, and numeric variables will remain unchanged.

The maximum number of characters that can be entered on an input line from the console is 127 by default. However, you may lower this limit by assigning an integer in the range 0 through 127 to the predefined variable *BufLen*.

Example:
```
Write('File name (max. 14 chars): ');
BufLen:=14;
Read(FileName);
```

Notice that assignments to *BufLen* affect only the immediately following *Read*. After that, *BufLen* is restored to 127.

Readln Procedure

The *Readln* procedure is identical to the *Read* procedure, except that after the last variable has been read, the remainder of the line is skipped. I.e., all characters up to and including the next CR/LF sequence (or the next CR on a logical device) are skipped. The syntax of the procedure statement is:


```
Readln(Var1,Var2,...,VarN)
```
or
```
Readln(FilVar,Var1,Var2,...,VarN)
```

After a *Readln*, the following *Read* or *Readln* will read from the beginning of the next line. *Readln* may also be called without parameters:

```
Readln
```
or
```
Readln(FilVar)
```

in which case the remaining of the line is skipped. When *Readln* is reading from the console (standard file *Input* or a file assigned to CON:), the terminating CR **is** echoed to the screen as a CR/LF sequence, as opposed to *Read*.

Write Procedure

The *Write* procedure provides output of characters, strings, boolean values, and numeric values. The syntax of a *Write* statement is:

Write(Var1,Var2,...,VarN)
or
Write(FilVar,Var1,Var2,...,VarN)

where *Var1, Var2,...,VarN* (the *write parameters*) are variables of type *Char, String, Boolean, Integer* or *Real*, optionally followed by a colon and an integer expression defining the width of the output field. In the first case, the variables are output to the the standard file *Output*, usually the screen. In the second case, the variables are output to the text file which is previously assigned to *FilVar*.

The format of a *write parameter* depends on the type of the variable. In the following descriptions of the different formats and their effects, the symbols:

I, m, n	denote *Integer* expressions,
R	denotes a *Real* expression,
Ch	denotes a *Char* expression,
S	denotes a *String* expression, and
B	denotes a *Boolean expression*.

ClibPDF - www.fastio.com

Write Parameters

Ch The character *Ch* is output.

Ch:n The character *Ch* is output right-adjusted in a field which is *n* characters wide, i.e. *Ch* is preceded by *n* − 1 blanks.

S The string *S* is output. Arrays of characters may also be output, as they are compatible with strings.

S:n The string *S* is output right-adjusted in a field which is *n* characters wide, i.e. *S* is preceded by *n* − *Length(S)* blanks.

B Depending on the value of *B*, either the word TRUE or the word FALSE is output.

B:n Depending on the value of *B*, either the word TRUE or the word FALSE is output right-adjusted in a field which is *n* characters wide.

I The decimal representation of the value of *I* is output.

I:n The decimal representation of the value of *I* is output right-adjusted in a field which is *n* characters wide.

R The decimal representation of the value of *R* is output in a field 18 characters wide, using floating point format. For *R* > = 0.0, the format is:

```
␣␣#.##########E*##
```

For R < 0.0, the format is:

```
␣-#.##########E*##
```

where ␣ represents a blank, # represents a digit, and * represents either plus or minus.

R:n The decimal representation of the value of *R* is output, right adjusted in a field *n* characters wide, using floating point format. For *R* > = 0.0:

```
blanks#.digitsE*##
```


For *R* < 0.0:

```
blanks-#.digitsE*##
```

where *blanks* represents zero or more blanks, *digits* represents from one to ten digits, # represents a digit, and * represents either plus or minus. As at least one digit is output after the decimal point, the field width is minimum 7 characters (8 for *R* < 0.0).

R:n:m The decimal representation of the value of *R* is output, right adjusted, in a field *n* characters wide, using fixed point format with *m* digits after the decimal point. No decimal part, and no decimal point, is output if *m* is 0. *m* must be in the range 0 through 24; otherwise floating point format is used. The number is preceded by an appropriate number of blanks to make the field width n.

Writeln Procedure

The *Writeln* procedure is identical to the *Write* procedure, except that a CR/LF sequence is output after the last value. The syntax of the *Writeln* statement is:

Writeln(Var1,Var1,Var2,...,VarN) or *Writeln(FilVar,Var1,Var2,...,VarN)*

A *Writeln* with no write parameters outputs an empty line consisting of a CR/LF sequence:

Writeln or *Writeln(FilVar)*

ClibPDF - www.fastio.com

Untyped Files

Untyped files are low-level I/O channels primarily used for direct access to any disk file using a record size of 128 bytes.

In input and output operations to untyped files, data is transferred directly between the disk file and the variable, thus saving the space required by the sector buffer required by typed files. An untyped file variable therefore occupies less memory than other file variables. As an untyped file is furthermore compatible with any file, the use of an untyped file is therefore to be preferred if a file variable is required only for *Erase*, *Rename* or other non-input/output operations.

An untyped file is declared with the reserved word **file**:

```
var
  DataFile: file;
```

BlockRead / BlockWrite

All standard file handling procedures and functions except *Read, Write,* and *Flush* are allowed on untyped files. *Read* and *Write* are replaced by two special high-speed transfer procedures: *BlockRead* and *BlockWrite*. The syntax of a call to these procedures is:

BlockRead(FilVar, Var, Recs)
BlockWrite(FilVar, Var, Recs)

or

BlockRead(FilVar, Var, Recs, Result)
BlockWrite(FilVar, Var, Recs, Result)

where *FilVar* variable identifier of an untyped file, *Var* is any variable, and *Recs* is an integer expression defining the number of 128-byte records to be transferred between the disk file and the variable. The **optional** parameter *Result* returns the number of records actually transferred.

The transfer starts with the first byte occupied by the variable *Var*. The programmer must insure that the variable *Var* occupies enough space to accommodate the entire data transfer. A call to *BlockRead* or *BlockWrite* also advances the file pointer *Recs* records.

A file to be operated on by *BlockRead* or *BlockWrite* must first be prepared by *Assign* and *Rewrite* or *Reset*. *Rewrite* creates and opens a new file, and *Reset* opens an existing file. After processing, *Close* should be used to ensure proper termination.

The standard function *EOF* works as with typed files. So do standard functions *FilePos* and *FileSize* and standard procedure *Seek*, using a component size of 128 bytes (the record size used by *BlockRead* and *BlockWrite*).

The following program uses untyped files to copy files of any type. Notice the use of the optional fourth parameter on *BlockRead* to check the number of records actually read from the source file.

```
program  FileCopy;
const
  RecSize      = 128;
  BufSize      = 200;
var
  Source, Dest: File;
  SourceName,
  DestName:      string[14];
  Buffer:        array[1..RecSize,1..BufSize] of Byte;
  RecsRead:      Integer;
begin
  Write('Copy from: ');
  Readln(SourceName);
  Assign(Source, SourceName);
  Reset(Source);
  Write('       To: ');
  Readln(DestName);
  Assign(Dest, DestName);
  Rewrite(Dest);
  repeat
    BlockRead(Source,Buffer,BufSize,RecsRead);
    BlockWrite(Dest,Buffer,RecsRead);
  until RecsRead = 0;
  Close(Source); Close(Dest);
end.
```


ClibPDF - www.fastio.com

I/O checking

The **I** compiler directive is used to control generation of runtime I/O error checking code. The default state is active, i.e. { $I + } which causes calls to an I/O check routine after each I/O operation. I/O errors then cause the program to terminate, and an error message indicating the type of error is displayed.

If I/O checking is passive, i.e. { $I − }, no run time checks are performed. An I/O error thus does not cause the program to stop, but suspends any further I/O until the standard function *IOresult* is called. When this is done, the error condition is reset and I/O may be performed again. It is now the programmer's responsibility to take proper action according to the type of I/O error. A zero returned by *IOresult* indicates a successful operation, anything else means that an error occurred during the last I/O operation. Appendix G lists all error messages and their Numbers. **Notice** that as the error condition is reset when IOresult is called, subsequent calls to IOresult will return zero until the next I/O error occurs.

The *IOresult* function is very convenient in situations where a program halt is an unacceptable result of an I/O error, like in the following example which continues to ask for a file name until the attempt to reset the file is successful (i.e. until an existing file name is entered):

```
procedure OpenInFile;
begin
  repeat
    Write('Enter name of input file ');
    Readln(InFileName);
    Assign(InFile, InFileName);
    {$I-} Reset(InFile) {$I+} ;
    OK := (IOresult = 0);
    if not OK then
      Writeln('Cannot find file ',InFileName);
  until OK;
end;
```


When the **I** directive is passive ({ $I-}), the following standard procedures should be followed by a check of IOresult to ensure proper error handling:

* Append	Close	Read	Seek
Assign	Erase	ReadLn	Write
BlockRead	Execute	Rename	WriteLn
BlockWrite	Flush	Reset	
Chain	* GetDir	Rewrite	
* ChDir	* MkDir	* RmDir	

* PC-DOS/MS-DOS only.

ClibPDF - www.fastio.com

Notes:

Chapter 15
POINTER TYPES

Variables discussed up to now have been *static*, i.e. their form and size is pre-determined, and they exist throughout the entire execution of the block in which they are declared. Programs, however, frequently need the use of a data structure which varies in form and size during execution. *Dynamic* variables serve this purpose as they are generated as the need arises and may be discarded after use.

Such dynamic variables are not declared in an explicit variable declaration like static variables, and they cannot be referenced directly by identifiers. Instead, a special variable containing the memory address of the variable is used to *point* to the variable. This special variable is called a *pointer variable*.

Defining a Pointer Variable

A pointer type is defined by the pointer symbol ^ succeeded by the type *identifier* of the dynamic variables which may be referenced by pointer variables of this type.

The following shows how to declare a record with associated pointers. The type *PersonPointer* is is declared as a *pointer to* variables of type *PersonRecord*:

```
type
  PersonPointer = ^PersonRecord;
  PersonRecord = record
                   Name: string[50];
                   Job: string[50];
                   Next: PersonPointer;
                 end;
Var
  FirstPerson, LastPerson, NewPerson: PersonPointer;
```

The variables *FirstPerson*, *LastPerson* and *NewPerson* are thus *pointer variables* which can point at records of type *PersonRecord*. As shown above, the type identifier in a pointer type definition may refer to an identifier which is not yet defined.

ClibPDF - www.fastio.com

Allocating Variables (New)

Before it makes any sense to use any of these pointer variables we must, of course, have some variables to point at. New variables of any type are allocated with the standard procedure *New*. The procedure has one parameter which must be a pointer to variables of the type we want to create.

A new variable of type *PersonRecord* can thus be created by the statement:

```
New(FirstPerson);
```

which has the effect of having *FirstPerson* point at a dynamically allocated record of type *PersonRecord*.

Assignments between pointer variables can be made as long as both pointers are of identical type. Pointers of identical type may also be compared using the relational operators = and <>, returning a *Boolean* result (*True* or *False*).

The pointer value **nil** is compatible with all pointer types. **nil** points to no dynamic variable, and may be assigned to pointer variables to indicate the absence of a usable pointer. **nil** may also be used in comparisons.

Variables created by the standard procedure *New* are stored in a stack-like structure called the *heap*. The TURBO Pascal system controls the heap by maintaining a heap pointer which at the beginning of a program is initialized to the address of the first free byte in memory. On each call to *New*, the heap pointer is moved towards the top of free memory the number of bytes corresponding to the size of the new dynamic variable.

Mark and Release

When a dynamic variable is no longer required by the program, the standard procedures *Mark* and *Release* are used to reclaim the memory allocated to these variables. The *Mark* procedure assigns the value of the heap pointer to a variable. The syntax of a call to *Mark* is:


```
Mark(Var);
```

where *Var* is a pointer variable. The *Release* procedure sets the heap pointer to the address contained in its argument. The syntax is:

```
Release(Var);
```

where *Var* is a pointer variable, previously set by *Mark*. *Release* thus discards *all* dynamic variables above this address, and cannot release the space used by variables in the middle of the heap. If you want to do that, you should use *Dispose* (see page 124) instead of *Mark/Release*.

The standard function *MemAvail* is available to determine the available space on the heap at any given time. Further discussion is deferred to chapters 20, 21, and 22.

ClibPDF - www.fastio.com

Using Pointers

Supposing we have used the *New* procedure to create a series of records of type *PersonRecord* (as in the example on the following page) and that the field *Next* in each record points at the next *PersonRecord* created, then the following statements will go through the list and write the contents of each record (*FirstPerson* points to the first person in the list):

```
while FirstPerson <> nil do
with FirstPerson^ do
begin
  Writeln(Name,' is a ',Job);
  FirstPerson := Next;
end;
```

FirstPerson^.Name may be read as *FirstPerson's.Name*, i.e. the field *Name* in the record pointed to by *FirstPerson*.

The following demonstrates the use of pointers to maintain a list of names and related job desires. Names and job desires will be read in until a blank name is entered. Then the entire list is printed. Finally, the memory used by the list is released for other use. The pointer variable *HeapTop* is used only for the purpose of recording and storing the initial value of the heap pointer. Its definition as a `^Integer` (pointer to integer) is thus totally arbitrary.


```
procedure Jobs;
type
  PersonPointer = ^PersonRecord;

  PersonRecord = record
                   Name: string[50];
                   Job: string[50];
                   Next: PersonPointer;
                 end;
Var
  HeapTop: ^Integer;
  FirstPerson, LastPerson, NewPerson: PersonPointer;
  Name: string[50];
begin
     FirstPerson := nil;
  Mark(HeapTop);
  repeat
    Write('Enter name:       ');
    Readln(Name);
    if Name <> '' then
    begin
      New(NewPerson);
      NewPerson^.Name := Name;
      Write('Enter profession: ');
      Readln(NewPerson^.Job);
      Writeln;
      if FirstPerson = nil then
        FirstPerson := NewPerson
      else
        LastPerson^.Next := NewPerson;
      LastPerson := NewPerson;
      LastPerson^.Next := nil;
    end;
  until Name='';
  Writeln;
  while FirstPerson <> nil do
  with FirstPerson^ do
  begin
     Writeln(Name,' is a ',Job);
     FirstPerson := Next;
  end;
  Release(HeapTop);
end.
```


ClibPDF - www.fastio.com

Dispose

Instead of *Mark* and *Release*, standard Pascal's *Dispose* procedure may be used to reclaim space on the heap.

NOTICE that *Dispose* and *Mark/Release* use entirely different approaches to heap management - **and never shall the twain meet!** Any one program must use **either** *Dispose* **or** *Mark/Release* to manage the heap. Mixing them will produce unpredictable results.

The syntax is:

```
Dispose(Var);
```

where *Var* is a pointer variable.

Dispose allows dynamic memory used *by a specific pointer variable* to be reclaimed for new use, as opposed to *Mark* and *Release* which releases the entire heap *from the specified pointer variable and upward.*

Suppose you have a number of variables which have been allocated on the heap. The following figure illustrates the contents of the heap and the effect of *Dispose(Var3)* and *Mark(Var3)/ Release(Var3)* :

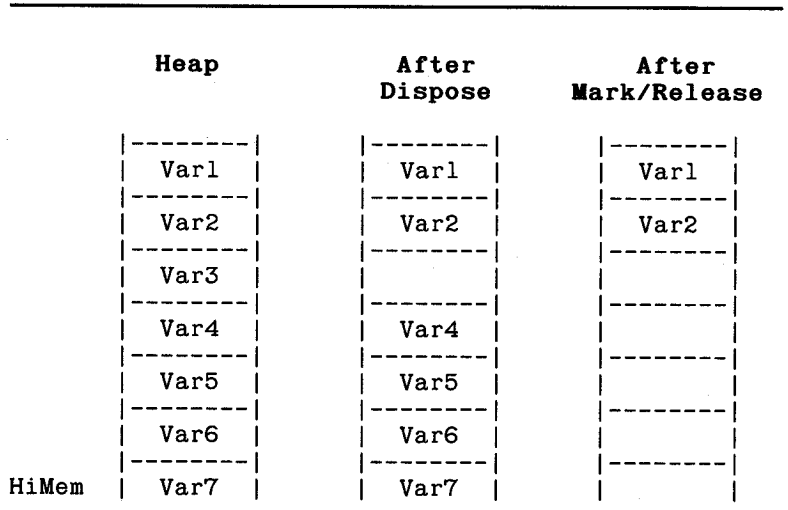

Figure 15-1: Using Dispose

After *Disposing* a pointer variable, the heap may thus consist of a number of memory areas in use interspersed by a number of free areas. Subsequent calls to *New* will use these if the new pointer variable fits into the space.

GetMem

The standard procedure *GetMem* is used to allocate space on the heap. Unlike *New*, which allocates as much space as required by the **type** pointed to by its argument, *GetMem* allows the programmer to control the amount of space allocated. *GetMem* is called with two parameters:

```
GetMem(PVar, I)
```

where *PVar* is any pointer variable, and *I* is an integer expression giving the number of bytes to be allocated.

FreeMem

Syntax: FreeMem;

The *FreeMem* standard procedure is used to reclaim an entire block of space on the heap. It is thus the counterpart of *GetMem*. *FreeMem* is called with two parameters:

```
FreeMem(PVar, I);
```

where *PVar* is any pointer variable, and *I* is an integer expression giving the number of bytes to be reclaimed, which must be **exactly** the number of bytes previously allocated to that variable by *GetMem*.

ClibPDF - www.fastio.com

MaxAvail

Syntax: MaxAvail;

The *MaxAvail* standard function returns the size of the largest consecutive block of free space on the heap. On 16-bit systems this space is in in number of *paragraphs* (16 bytes each); on 8-bit systems it is in bytes. The result is an *Integer*, and if more than 32767 paragraphs/bytes are available, *MaxAvail* returns a negative number. The correct number of free paragraphs/bytes is then calculated as 65536.0 + *MaxAvail*. Notice the use of a real constant to generate a *Real* result, as the result is greater than *MaxInt*.

Hints

Note that no range checking is done on pointers. It is the responsibility of the programmer to ensure that a pointer points to a legal address.

If you have difficulties using pointers, a drawing of what you are tempting to do often clears up things.

Chapter 16
PROCEDURES AND FUNCTIONS

A Pascal program consists of one or more *blocks*, each of which may again consist of blocks, etc. One such block is a *procedure*, another is a *function* (in common called *subprograms*). Thus, a procedure is a separate part of a program, and it is activated from elsewhere in the program by a *procedure statement* (see page 56). A function is rather similar, but it computes and returns a value when its identifier, or *designator*, is encountered during execution (see page 54).

Parameters

Values may be passed to procedures and functions through *parameters*. Parameters provide a substitution mechanism which allows the logic of the subprogram to be used with different initial values, thus producing different results.

The procedure statement or function designator which invokes the subprogram may contain a list of parameters, called the *actual parameters*. These are passed to the *formal parameters* specified in the subprogram *heading*. The order of parameter passing is the order of appearance in the parameter lists. Pascal supports two different methods of parameter passing: by *value* and by *reference*, which determines the effect that changes of the formal parameters have on the actual parameters.

When parameters are passed *by value*, the formal parameter represents a local variable in the subprogram, and changes of the formal parameters have no effect on the actual parameter. The actual parameter may be any expression, including a variable, with the same type as the corresponding formal parameter. Such parameters are called a *value parameter* and are declared in the subprogram heading as in the following example. This and the following examples show procedure headings; see page 137 for a description of function headings.

```
procedure Example(Num1,Num2: Number; Str1,Str2: Txt);
```


ClibPDF - www.fastio.com

Number and *Txt* are previously defined types (e.g. *Integer* and **string**[255]), and *Num1, Num2, Str1,* and *Str2* are the *formal parameters* to which the value of the *actual parameters* are passed. The types of the formal and the actual parameters must correspond.

Notice that the type of the parameters in the parameter part must be specified as a previously defined *type identifier*. Thus, the construct:

```
procedure Select(Model: array[1..500] of Integer);
```

is **not** allowed. Instead, the desired type should be defined in the **type** definition of the block, and the *type identifier* should then be used in the parameter declaration:

```
type
  Range = array[1..500] of Integer;

procedure Select(Model: Range);
```

When a parameter is passed *by reference*, the formal parameter in fact represents the actual parameter throughout the execution of the subprogram. Any changes made to the formal parameter is thus made to the actual parameter, which must therefore be a *variable*. Parameters passed by reference are called a *variable parameters*, and are declared as follows:

```
procedure Example(Var Num1,Num2: Number)
```

Value parameters and variable parameters may be mixed in the same procedure as in the following example:

```
procedure Example(Var Num1,Num2: Number; Str1,Str2: Txt);
```

in which *Num1* and *Num2* are variable parameters and *Str1* and *Str2* are value parameters.

All address calculations are done at the time of the procedure call. Thus, if a variable is a component of an array, its index expression(s) are evaluated when the subprogram is called.

Notice that **file** parameters must always be declared as variable parameters.

When a large data structure, such as an array, is to be passed to a subprogram as a parameter, the use of a variable parameter will save both time and storage space, as the only information then passed on to the subprogram is the address of the actual parameter. A value parameter would require storage for an extra copy of the entire data structure, and the time involved in copying it.

Relaxations on Parameter Type Checking

Normally, when using variable parameters, the formal and the actual parameters must match exactly. This means that subprograms employing variable parameters of type *String* will accept only strings of the exact length defined in the subprogram. This restriction may be overridden by the **V** compiler directive. The default active state { $V+ } indicates strict type checking, whereas the passive state { $V- } relaxes the type checking and allows actual parameters of any string length to be passed, irrespective of the length of the formal parameters.

Example:

```
program Encoder;
{$V-}
type
  WorkString = string[255];
Var
  Line1: string[80];
  Line2: string[100];
procedure Encode(Var LineToEncode: WorkString);
Var I: Integer;
begin
  for I := 1 to Length(LineToEncode) do
    LinetoEncode[I] := Chr(Ord(LineToEncode[I])-30);
end;
begin
  Line1 := 'This is a secret message';
  Encode(Line1);
  Line2 := 'Here is another (longer) secret message';
  Encode(Line2);
end.
```


ClibPDF - www.fastio.com

Untyped Variable Parameters

If the type of a formal parameter is not defined, i.e. the type definition is omitted from the parameter section of the subprogram heading, then that parameter is said to be *untyped*. Thus, the corresponding actual parameter may be any type.

The untyped formal parameter itself is incompatible with all types, and may be used only in contexts where the data type is of no significance, for example as a parameter to *Addr, BlockRead/Write, FillChar,* or *Move*, or as the address specification of **absolute** variables.

The *SwitchVar* procedure in the following example demonstrates the use of untyped parameters. It moves the contents of the variable *A1* to *A2* and the contents of *A2* to *A1*.

```
procedure SwitchVar(Var Alp,A2p; Size: Integer);
type
  A = array[1..MaxInt] of Byte;
Var
  A1: A absolute Alp;
  A2: A absolute A2p;
  Tmp: Byte;
  Count: Integer;
begin
  for Count := 1 to Size do
  begin
    Tmp := A1[Count];
    A1[Count] := A2[Count];
    A2[Count] := Tmp;
  end;
end;
```

Assuming the declarations:

```
type
  Matrix = array[1..50,1..25] of Real;
Var
  TestMatrix,BestMatrix:  Matrix;
```

then *SwitchVar* may be used to switch values between the two matrices:

```
SwitchVar(TestMatrix,BestMatrix, SizeOf(Matrix));
```


Procedures

A procedure may be either pre-declared (or 'standard') or user-declared, i.e. declared by the programmer. Pre-declared procedures are parts of the TURBO Pascal system and may be called with no further declaration. A user-declared procedure may be given the name of a standard procedure; but that standard procedure then becomes inaccessible within the scope of the user declared procedure.

Procedure Declaration

A procedure declaration consists of a procedure heading followed by a block which consists of a declaration part and a statement part.

The procedure heading consists of the reserved word **procedure** followed by an identifier which becomes the name of the procedure, optionally followed by a formal parameter list as described on page 127 .

Examples:

```
procedure LogOn;
procedure Position(X,Y: Integer);
procedure Compute(Var Data: Matrix; Scale: Real);
```

The declaration part of a procedure has the same form as that of a program. All identifiers declared in the formal parameter list and the declaration part are local to that procedure, and to any procedures within it. This is called the *scope* of an identifier, outside which they are not known. A procedure may reference any constant, type, variable, procedure, or function defined in an outer block.

The statement part specifies the action to be executed when the the procedure is invoked, and it takes the form of a compound statement (see page 57). If the procedure identifier is used within the statement part of the procedure itself, the procedure will execute recursively. (**CP/M-80 only:** Notice that the **A** compiler directive must be passive { $A-} when recursion is used, see Appendix C.)

The next example shows a program which uses a procedure and passes a parameter to this procedure. As the actual parameter passed to the procedure is in some instances a constant (a simple expression), the formal parameter must be a value parameter.

ClibPDF - www.fastio.com

```
program Box;
Var
  I: Integer;
procedure DrawBox(X1,Y1,X2,Y2: Integer);
  Var I: Integer;
  begin
     GotoXY(X1,Y1);
     for I := X1 to X2 do write('-');
     for I := Y1+1 to Y2 do
     begin
       GotoXY(X1,I); Write('!');
       GotoXY(X2,I); Write('!');
     end;
     GotoXY(X1,Y2);
     for I := X1 to X2 do Write('-');
  end; { of procedure DrawBox }
begin
  ClrScr;
  for I := 1 to 5 do DrawBox(I*4,I*2,10*I,4*I);
  DrawBox(1,1,80,23);
end.
```

Often the changes made to the formal parameters in the procedure should also affect the actual parameters. In such cases *variable parameters* are used, as in the following example:

```
procedure Switch(Var A,B: Integer);
Var Tmp: Integer;
begin
  Tmp := A; A := B; B := Tmp;
end;
```

When this procedure is called by the statement:

```
Switch(I,J);
```

the values of **I** and **J** will be switched. If the procedure heading in **Switch** was declared as:

```
procedure Switch(A,B: Integer);
```

i.e. with a *value* parameter, then the statement `Switch(I,J)` would **not** change *I* and *J*.

Standard Procedures

TURBO Pascal contains a number of standard procedures. These are:

1) string handling procedures (described on pages 71 pp),
2) file handling procedures (described on pages 94, 101, and 114).
3) procedures for allocation of dynamic variables (described on pages 120 and 125), and
4) input and output procedures (described on pages 108 pp).

In addition to these, the following standard procedures are available, provided that the associated commands have been installed for your terminal (see pages 12 pp):

ClrEol

Syntax: ClrEol;

Clears all characters from the cursor position to the end of the line without moving the cursor.

ClrScr

Syntax: ClrScr;

Clears the screen and places the cursor in the upper left-hand corner. Beware that some screens also reset the video-attributes when clearing the screen, possibly disturbing any user-set attributes.

CrtInit

Syntax: CrtInit;

Sends the *Terminal Initialization String* defined in the installation procedure to the screen.

ClibPDF - www.fastio.com

CrtExit

Syntax: CrtExit;

Sends the *Terminal Reset String* defined in the installation procedure to the screen.

Delay

Syntax: Delay(*Time*);

The *Delay* procedure creates a loop which runs for approx. as many milliseconds as defined by its argument *Time* which must be an integer. The exact time may vary somewhat in different operating environments.

DelLine

Syntax: DelLine;

Deletes the line containing the cursor and moves all lines below one line up.

InsLine

Syntax: InsLine;

Inserts an empty line at the cursor position. All lines below are moved one line down and the bottom line scrolls off the screen.

GotoXY

Syntax: GotoXY(*Xpos,Ypos*);

Moves the cursor to the position on the screen specified by the integer expressions *Xpos* (horizontal value, or *row*) and *Ypos* (vertical value, or *column*). The upper left corner (home position) is (1,1).

Exit

Syntax: Exit;

Exits the current block. When exit is executed in a subroutine, it causes the subroutine to return. When it is executed in the statement part of a program, it causes the program to terminate. A call to *Exit* may be compared to a **goto** statement addressing a label just before the **end** of a block.

Halt

Syntax: Halt;

Stops program execution and returns to the operating system.

In PC/MS-DOS, *Halt* may optionally pass a integer parameter specifying the return code of the program. *Halt* without a parameter corresponds to *Halt(O).* The return code may be examined by the parent process using an MS-DOS system function call or through an ERRORLEVEL test in an MS-DOS batch file.

LowVideo

Syntax: LowVideo;

Sets the screen to the video attribute defined as 'Start of Low Video' in the installation procedure, i.e. 'dim' characters.

NormVideo

Syntax: NormVideo;

Sets the screen to the video attribute defined as 'Start of Normal Video' in the installation procedure, i.e. the 'normal' screen mode.

Randomize

Syntax: Randomize;

Initializes the random number generator with a random value.

ClibPDF - www.fastio.com

Move

Syntax: Move(*var1,var2,Num*);

Does a mass copy directly in memory of a specified number of bytes. *var1* and *var2* are two variables of any type, and *Num* is an integer expression. The procedure copies a block of *Num* bytes, starting at the first byte occupied by *var1* to the block starting at the first byte occupied by *var2*. You may notice the absence of explicit 'moveright' and 'moveleft' procedures. This is because *Move* automatically handles possible overlap during the move process.

FillChar

Syntax: FillChar(*Var, Num, Value*);

Fills a range of memory with a given value. *Var* is a variable of any type, *Num* is an integer expression, and *Value* is an expression of type *Byte* or *Char*. *Num* bytes, starting at the first byte occupied by *Var*, are filled with the value *Value*.

Functions

Like procedures, functions are either standard (pre-declared) or declared by the programmer.

Function Declaration

A function declaration consists of a function *heading* and a *block* which is a declaration part followed by a statement part.

The function heading is equivalent to the procedure heading, except that the heading must define the *type* of the function result. This is done by adding a colon and a type to the heading as shown here:

```
function KeyHit: Boolean;
function Compute(Var Value: Sample): Real;
function Power(X,Y: Real): Real;
```

The result type of a function must be a scalar type (i.e. *Integer, Real, Boolean, Char,* declared scalar or subrange), a **string** type, or a pointer type.

The declaration part of a function is the same as that of a procedure.

The statement part of a function is a compound statement as described on page 57 . Within the statement part at least one statement assigning a value to the function identifier must occur. The last assignment executed determines the result of the function. If the function designator appears in the statement part of the function itself, the function will be invoked recursively. (**CP/M-80 only:** Notice that the **A** compiler directive must be passive { $A-} when recursion is used, see Appendix C.)

ClibPDF - www.fastio.com

The following example shows the use of a function to compute the sum of a row of integers from **I** to **J**.

```
function RowSum(I,J: Integer): Integer;
  function SimpleRowSum(S: Integer): Integer;
  begin
    SimpleRowSum := S*(S+1) div 2;
  end;
begin
  RowSum := SimpleRowSum(J)-SimpleRowSum(I-1);
end;
```

The function *SimpleRowSum* is nested within the function *RowSum*. *SimpleRowSum* is therefore only available within the scope of *RowSum*.

The following program is the classical demonstration of the use of a recursive function to calculate the factorial of an integer number:

```
{$A-} {A- directive allows recursion in CP/M-80 version}
program Factorial;
Var Number: Integer;
function Factorial(Value: Integer): Real;
begin
  if Value = 0  then Factorial := 1
  else Factorial := Value * Factorial(Value-1);
end;
begin
  Read(Number);
  Writeln(^M,Number,'! = ',Factorial(Number));
end.
```

Note that the type used in the definition of a function type must be previously specified as a *type identifier*. Thus, the construct:

```
function LowCase(Line: UserLine): string[80];
```

is **not** allowed. Instead, a type identifier should be associated with the type *string[80]*, and that type identifier should then be used to define the function result type, for example:

```
type
  Str80 = string[80];

function LowCase(Line: UserLine): Str80;
```


Because of the implementation of the standard procedures *Write* and *Writeln*, a function using any of the standard procedures *Read, Readln, Write,* or *Writeln,* must **never** be called by an expression within a *Write* or *Writeln* statement. In 8-bit systems this is also true for the standard procedures *Str* and *Val*.

Standard Functions

The following standard (pre-declared) functions are implemented in TURBO Pascal:

1) string handling functions (described on pages 71 pp),
2) file handling functions (described on pages 94 and 101),
3) pointer related functions (described on pages 120 and 125).

Arithmetic Functions

Abs

Syntax: Abs(*Num*);

Returns the absolute value of *Num*. The argument *Num* must be either *Real* or *Integer*, and the result is of the same type as the argument.

ArcTan

Syntax: ArcTan(*Num*);

Returns the angle, in radians, whose tangent is *Num*. The argument *X* must be either *Real* or *Integer*, and the result is *Real*.

Cos

Syntax: Cos(*Num*);

Returns the cosine of *Num*. The argument *Num* is expressed in radians, and its type must be either *Real* or *Integer*. The result is of type *Real*.

ClibPDF - www.fastio.com

Exp

Syntax: Exp(*Num*);

Returns the exponential of *Num*, i.e. *enum*. The argument *Num* must be either *Real* or *Integer*, and the result is *Real*.

Frac

Syntax: Frac(*Num*);

Returns the fractional part of *Num*, i.e. Frac(*Num*) = *Num* - Int(*Num*). The argument *Num* must be either *Real* or *Integer*, and the result is *Real*.

Int

Syntax: Int(*Num*);

Returns the integer part of *Num*, i.e. the greatest integer number less than or equal to *Num*, if *Num* > = 0, or the smallest integer number greater than or equal to *Num*, if *Num* < 0. The argument *Num* must be either *Real* or *Integer*, and the result is *Real*.

Ln

Syntax: Ln(*Num*);

Returns the natural logarithm of *Num*. The argument *Num* must be either *Real* or *Integer*, and the result is *Real*.

Sin

Syntax: Sin(*Num*);

Returns the sine of *Num*. The argument *Num* is expressed in radians, and its type must be either *Real* or *Integer*. The result is of type *Real*.

Sqr

Syntax: Sqr(*Num*);

Returns the square of *Num*, i.e. *Num* * *Num*. The argument *Num* must be either *Real* or *Integer*, and the result is of the same type as the argument.

Sqrt

Syntax: Sqrt(*Num*);

Returns the square root of *Num*. The argument *Num* must be either *Real* or *Integer*, and the result is *Real*.

Scalar Functions

Pred

Syntax: Pred(*Num*);

Returns the predecessor of *Num* (if it exists). *Num* is of any scalar type.

Succ

Syntax: Succ(*Num*);

Returns the successor of *Num* (if it exists). *Num* is of any scalar type.

Odd

Syntax: Odd(*Num*);

Returns boolean *True* is *Num* is an odd number, and *False* if *Num* is even. *Num* must be of type *Integer*.

ClibPDF - www.fastio.com

Transfer Functions

The transfer functions are used to convert values of one scalar type to that of another scalar type. In addition to the following functions, the *retype* facility described on page 65 serves this purpose.

Chr

Syntax: Chr(*Num*);

Returns the character with the ordinal value given by the integer expression *Num*. Example: *Chr(65)* returns the character 'A'.

Ord

Syntax: Ord(*Var*);

Returns the ordinal number of the value *Var* in the set defined by the type *Var*. *Ord(Var)* is equivalent to *Integer(Var)* (see Type Conversions on page 65. *Var* may be of any scalar type, except *Real*, and the result is of type *Integer*.

Round

Syntax: Round(*Num*);

Returns the value of *Num* rounded to the nearest integer as follows: if *Num* > = 0, then *Round(Num)* = *Trunc(Num + 0.5)*, and if *Num* < 0, then *Round(Num)* = *Trunc(Num − 0.5) Num* must be of type *Real*, and the result is of type *Integer*.

Trunc

Syntax: Trunc(*Num*);

Returns the greatest integer less than or equal to *Num*, if *Num* > = 0, or the smallest integer greater than or equal to *Num*, if *Num* < 0. *Num* must be of type *Real*, and the result is of type *Integer*.

Miscellaneous Standard Functions

Hi

Syntax: Hi(*I*);

The low order byte of the result contains the high order byte of the value of the integer expression *I*. The high order byte of the result is zero. The type of the result is *Integer*.

KeyPressed

Syntax: KeyPressed

Returns boolean *True* if a key has been pressed at the console, and *False* if no key has been pressed. The result is obtained by calling the operating system console status routine.

Lo

Syntax: Lo(*I*);

Returns the low order byte of the value of the integer expression *I* with the high order byte forced to zero. The type of the result is *Integer*.

Random

Syntax: Random;

Returns a random number greater than or equal to zero and less than one. The type is *Real*.

Random(Num)

Syntax: Random(*Num*);

Returns a random number greater than or equal to zero and less than *Num*. *Num* and the random number are both *Integers*.

ClibPDF - www.fastio.com

ParamCount

Syntax: ParamCount;

This integer function returns the number of parameters passed to the program in the command line buffer. Space and tab characters serve as separators.

ParamStr

Syntax: ParamStr(*N*);

This string function returns the *N*'th parameter from the command line buffer.

SizeOf

Syntax: SizeOf(*Name*);

Returns the number of bytes occupied in memory by the variable or type *Name*. The result is of type *Integer*.

Swap

Syntax: Swap(*Num*);

The Swap function exchanges the high and low order bytes of its integer argument *Num* and returns the resulting value as an integer.

Example:
`Swap($1234)` returns *$3412* (values in hex for clarity).

UpCase

Syntax: UpCase(*ch*);

Returns the uppercase equivalent of its argument *ch* which must be of type *Char*. If no uppercase equivalent exists, the argument is returned unchanged.

Forward References

A subprogram is **forward** declared by specifying its heading separately from the block. This separate subprogram heading is exactly as the normal heading, except that it is terminated by the reserved word **forward**. The block follows later within the same declaration part. Notice that the block is initiated by a copy of the heading, specifying only the name and no parameters, types, etc.

Example:

```
program Catch22;
Var
  X: Integer;
function Up(Var I: Integer): Integer; forward;
function Down(Var I: Integer): Integer;
begin
  I := I div 2; Writeln(I);
  if I <> 1 then I := Up(I);
end;
function Up;
begin
  while I mod 2 <> 0  do
  begin
    I := I*3+1; Writeln(I);
  end;
  I := Down(I);
end;
begin
  Write('Enter any integer: ');
  Readln(X);
  X := Up(X);
  Write('Ok. Program stopped again.');
end.
```

When the program is executed and if you enter e.g. *6* it outputs:

CibPDF - www.fastio.com

```
3
10
5
16
8
4
2
1
Ok. Program stopped again.
```

The above program is actually a more complicated version of the following program:

```
program Catch222;
Var
  X: Integer;
begin
  Write('Enter any integer: ');
  Readln(X);
  while X <> 1 do
  begin
    if X mod 2 = 0 then X := X div 2 else X := X*3+1;
    Writeln(X);
  end;
  Write('Ok. Program stopped again.');
end.
```

It may interest you to know that it cannot be proved if this small and very simple program actually **will** stop for any integer!

Chapter 17
INCLUDING FILES

The fact that the TURBO editor performs editing only within memory limits the size of source code handled by the editor. The **I** compiler directive can be used to circumvent this restriction, as it provides the ability to split the source code into smaller 'lumps' and put it back together at compile-time. The include facility also aids program clarity, as commonly used subprograms, once tested and debugged, may be kept as a 'library' of files from which the necessary files can be included in any other program.

The syntax for the **I** compiler directive is:

```
{$I filename}
```

where *filename* is any legal file name. Leading spaces are ignored and lower case letters are translated to upper case. If no file type is specified, the default type **.PAS** is assumed. This directive must be specified on a line by itself.

Examples:

```
{$Ifirst.pas}
{$I COMPUTE.MOD}
{$iStdProc }
```

Notice that a space must be left between the file name and the closing brace if the file does not have a three-letter extension; otherwise the brace will be taken as part of the name.

To demonstrate the use of the include facility, let us assume that in your 'library' of commonly used procedures and functions you have a file called *STUPCASE.FUN*. It contains the function *StUpCase* which is called with a character or a string as parameter and returns the value of this parameter with any lower case letters set to upper case.

ClibPDF - www.fastio.com