String handling routines

Michaël Van Canneyt

February 6, 2010

Abstract

Pascal has long been known for its easy string manipulations. With the coming of dynamically typed languages such as Python, Perl, PHP, Javascript, this advantage has somewhat lessened, as these languages make manipulating strings also less cumbersome. If well programmed, Pascal programs have still the advantage: Speed (because it is compiled) and safe (type safety). An overview.

1 Introduction

Many - if not all - computer programs manipulate text (strings) at some point, if just to show a message on the screen. The ease of use of string handling is therefor an important asset. In the realm of compiled languages, Pascal has traditionally held the advantage over C, due to the ease of use and the type safety - inherently making a pascal program more safe from buffer overflows than a C program ([see article on strings sent by Rosa]). This inherent unsafety is probably the reason why C is rarely - if at all - is used for webserver programming, favouring interpreted, dynamically typed, languages.

Security issues aside, Pascal has from the very start provided a set of intuitive routines to manipulate strings. Some of these have been incorporated in the system unit, so they are available in all Pascal programs. A large part of the routines has been put in the SysUtils and later the StrUtils units. In this article an overview of what is available is presented.

2 A word about string types

In the beginning days of Pascal, strings were limited to a length of 255 characters, each character 1 byte long. Strings of this type are very efficient in terms of manipulation and speed: they don't use heap memory. This string type still exists today, and is called `ShortString`.

The 255 character limit became increasingly a problem, so a new string type was introduced that lifted this limit: the `AnsiString` type. A small exception aside, it could be used just as a shortstring. This string was efficient in memory due to an inherent reference counting mechanism, but slower in usage, since it required heap memory, plus extra exception handling mechanisms: all this is hidden from the programmer, but has a performance impact.

Both string types suffered from the problem that they worked with single-byte characters, which could contain only characters of 1 codepage (at most 255 different characters). So the Widestring type was introduced, which sported 2 bytes per character, and was able to support all unicode characters. This type was not reference counted, but was used in windows OLE (activeX) mechanisms. This string type was much less peformant as ansistrings:

no reference counting. Additionally, special Windows memory management is needed to allocate memory for them. This string type string requires still a lot of manipulations when codepage conversions need to be performed or when they must be converted to the easier AnsiString type.

Delphi 2009 introduced a new string type: it works like an ansistring, but has additionally a codepage associated with it (Free Pascal support for this string type is forthcoming). Codepage conversions are handled automatically. Other than that, they work just like ansistrings. This additional functionality comes again with a price: The actual size (in bytes) of a characters is unknown, and the checking and optional converting of code pages requires additional time.

Nevertheless, all what follows should be independent of the actual string type used.

3 Basic operations

There are 6 basic operations that can be performed on a piece of text: finding its length, putting 2 pieces together, taking a piece out of it, deleting a part of the string, inserting something in the string, or searching for a word. These operations are summarized in 6 routines of the system unit (in the same order):

```
// Return the length of a string
function Length(S : string): Integer;
// Append S2 to S1 and return result
function Concat(s1: string; s2: string): string
// Return count characters from S starting at Index
function Copy(S : String;
              Index: Integer; Count: Integer): string;
// Delete Count caracters from S starting at Index
procedure Delete(var S: string;
                 Index: Integer; Count: Integer);
// insert Source in S at position Index
procedure Insert(Source: string;
                 var S: string; Index: Integer);
// return (1-based) position of Substr in S.
function Pos(const substr: AnsiString;
             const str: AnsiString): Integer;
```

The `Concat` operation is a left-over from the beginning days of Pascal: the + operator performs the same task. In fact, `Concat` can be written as:

```
function Concat(s1: string; s2: string): string;
begin
  Result:=S1+S2;
end;
```

The `Insert` operation can actually be expressed as a `Copy` operation:

```
procedure Insert(Src: string;
                 var S: string; Index: Integer);
begin
  S:=Copy(S,1,Index-1)+Src+Copy(S,Index,Length(S)-Index+1);
end;
```

Note that event his implementation is safe (handles bad situations): the `Copy` function will do all necessary checks on valid values for `Index`.

With these 6 routines, an amazing number of things can be done. For instance, the following search-and-replace routine uses almost all the basic routines:

```
Function replace(Const AText, Src,Dest : String) : String;

Var
  P : Integer;
  S : String;

begin
  Result:='';
  S:=AText;
  // While there is any text to search left
  While (Length(S)>0) do
    begin
// search for text
    P:=Pos(Src,S);
    If P=0 then
      begin
      // nothing found, copy rest to result
      Result:=Result+S;
      S:='';
      end
    else
      begin
      // Append text prior to match, and append dest
      Result:=Result+Copy(S,1,P-1)+Dest;
      // Delete match from text to search.
      Delete(S,1,P+Length(Src)-1);
      end;
    end;
end;
```

The above code is an example of how easy the basic functions work. The below screenshot shows how it can be used in practice:

The `OnClick` handler of the 'Go' button looks as follows:

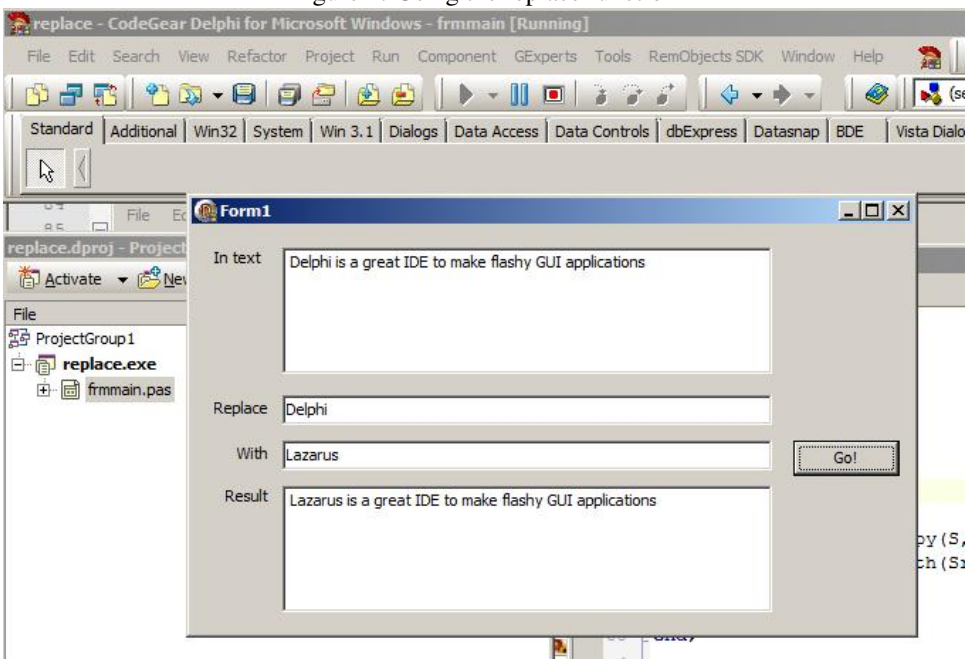
```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MResult.Text:=Replace(MText.Text,ESrc.Text,EDest.Text);
end;
```

Nothing could be easier.

4 Case sensitivity

Pascal is a case-insensitive language. That is, it does not matter if one writes 'Begin' or 'begin'. The standard string routines and operations, however, are case sensitive. The following expressions will return False:

Figure 1: Using the replace function




```
'Begin'='begin'
Pos('delphi','Delphi is a RAD tool')=1;
```

The system unit has (for historic reasons, it existed in Turbo Pascal) only 1 function to deal with the case of strings:

```
Function UpCase(C : Char) : Char;
```

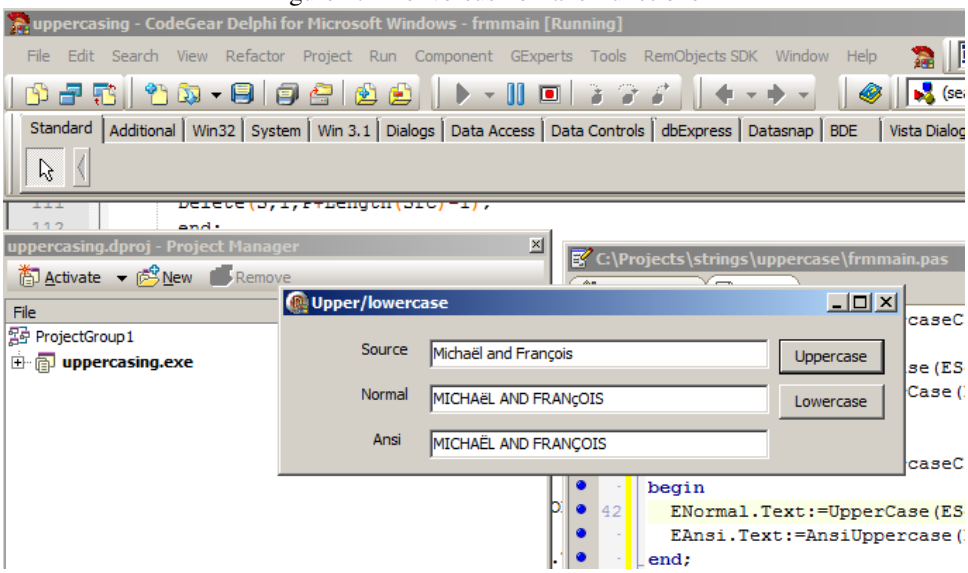
It converts a single letter in the range 'a'..'z' to its uppercase equivalent. It is obvious that this is a very limited function.

Luckily, the `SysUtils` unit contains more functions to handle case sensitivity:

```
// Convert normal letters to uppercase
function UpperCase(S: string): string;
// Convert all letters to uppercase
function AnsiUpperCase(S: string): string;
// Convert normal letters to lowercase
function LowerCase(: string): string;
// Convert all letters to lowercase
function AnsiLowerCase(S: string): string;
// Compare 2 strings, ignoring case for 'a'..'z'
function CompareText(S1, S2: string): Integer;
// Compare 2 strings, ignoring case
function AnsiCompareText(S1, S2: string): Integer;
```

The compare functions return an integer result. Zero indicates the two strings are equal, less than zero means S1 is less than S2, larger than zero means S1 is larger than S2. The following screenshot demonstrates the differences between the working of the `Uppercase` and `AnsiUppercase` function: The same holds for the lowercase and compare functions. The `OnClick` handler of the 'Uppercase' button looks as follows:

Figure 2: Ansi versus non-ansi functions




```
procedure TForm1.BUppercaseClick(Sender: TObject);
begin
  ENormal.Text:=UpperCase(ESource.Text);
  EAnsi.Text:=AnsiUppercase(ESource.Text);
end;
```

The `Comparetext` and `AnsiCompareText` will give different results when comparing the 2 resulting strings. `CompareText` will indicate that they are different, while `AnsiCompareText` will declare them to be equal, because `CompareText` will compare all letters not in the regular alphabet based on their byte values.

There are some wrapper functions around `CompareText`:

```
// Check if 2 strings are equal, ignoring case for 'a'..'z'
function SameText(const S1, S2: string): Boolean;
// Check if 2 strings are equal, ignoring case
function AnsiSameText(const S1, S2: string): Boolean;
```

They simply check whether the `CompareText` and `AnsiCompareText` functions return zero, and return `True` if this is the case.

The `Compare*` and `Same*` functions exist in variants that compare strings case sensitively. Instead of ending on 'Text', their names end on 'Str':

```
// Compare 2 strings
function CompareStr(S1, S2: string): Integer;
// Compare 2 strings, in current locale
function AnsiCompareStr(S1, S2: string): Integer;
// Check if 2 strings are equal
function SameStr(const S1, S2: string): Boolean;
// Check if 2 strings are equal in current locale
function AnsiSameStr(const S1, S2: string): Boolean;
```

The difference between the Ansi and non-Ansi version lies in the way they treat special characters such as á and à: the non-ansi versions will compare them according to their

ASCII values. This can be tested with the comparing program, provided on the disc accompagnying this issue.

5 Searching and replacing

The standard `Pos` function can be used for rudimentary searching. The `AnsiPos` function of the `SysUtils` unit does the same thing, only it takes care of multi-byte characters, as used in the Far East. There is no case-insensitive version of `Pos`, but the following 2 functions from the StrUtils unit can be used to detect the presence of one string as part of another:

```
// Return True if AText contains ASubText, case sensitive
function AnsiContainsStr(Const AText: string;
                         Const ASubText: string): Boolean;
// Return True if AText contains ASubText, case insensitive
function AnsiContainsText(const AText: string;
                          const ASubText: string): Boolean;
```

These functions do not return the exact position, just the presence. The `StrUtils` unit does contain two other extensions to `Pos`. The first is called `PosEx`:

```
function PosEx(Const SubStr: string;
               Const S: string;
   Offset: Integer = 1): Integer;
```

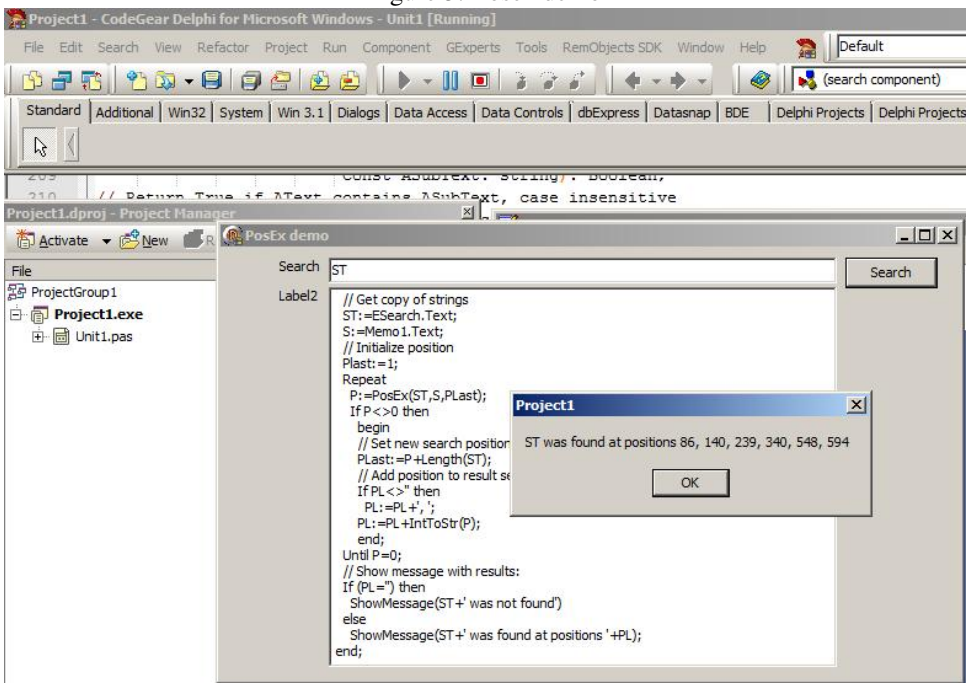
It functions the same as `Pos`, but instead of starting the search at the first character of the string `S`, it starts at position `Offset`. It can be used to search for multiple matches in a string by continuing the search at the last match:

```
procedure TForm1.BSearchClick(Sender: TObject);

Var
  Plast,P : Integer;
  S,ST,PL : String;

begin
  // Get copy of strings
  ST:=ESearch.Text;
  S:=Memo1.Text;
  // Initialize position
  Plast:=1;
  Repeat
    P:=PosEx(ST,S,PLast);
    If P<>0 then
      begin
      // Set new search position
      PLast:=P+Length(ST.Text);
      // Add position to result set
      If PL<>'' then
        PL:=PL+', ';
      PL:=PL+IntToStr(P);
      end;
  Until P=0;
```

Figure 3: Posex demo




```
  // Show message with results:
  If (PL='') then
    ShowMessage(ST+' was not found')
  else
    ShowMessage(ST+' was found at positions '+PL);
end;
```

The result is shown in figure 3 on page 7.

An even more extended version of `Pos` exists in `StrUtils`, and is called `SearchBuf`:

```
function SearchBuf(Buf: PChar; BufLen: Integer;
                   SelStart: Integer; SelLength: Integer;
   SearchStrin g: String;
   Options: TStringSearchOptions): PChar;
```

It searchs the `Buf` null-terminated string buffer for `SearchString`. A set of `SelStart`, `SelLength` indicators must be used to specify a search area (note that `SelStart` is 0-based, as common with null-terminated strings). The optional `Options` parameter can be used to further control the search:

soDown Search forward (the default) or backward from the end of the selection. If this option is not specified, `Selstart` is the starting position of the search (from which the search goes to the beginning of the buffer), and `SelLength` is ignored.

soMatchCase Search case sensitive.

soWholeWord Match only whole words. Words are any combination of alphabetical and numerical characters, all other characters denote non-words. This is a problem, as accented characters for instance will be considered whitespace.

The function works on a null-terminated string, and returns a null-terminated string.

This is not a very safe, but the following function is:

```
function SearchBuf(Buf : String;
                   SelStart, SelLength: Integer;
                   SearchString: String;
                   Options: TStringSearchOptions): Integer;

Var
  PR,PBuf : PChar;

begin
  Result:=0;
  If (Buf<>'') then
    begin
    PBuf:=PChar(Buf);
    PR:=strutils.SearchBuf(PBuf,Length(Buf),
                           SelStart-1,SelLength,
                           SearchString,Options);
    If (PR<>Nil) then
      Result:=(PR-PBuf)+1;
    end;
end;
```

It accepts a regular pascal string, the `SelStart` parameter is 1-based, and the return value is the position in the string (1 based), as in the `Pos` function.

The function is easier to use than the one in sysutils. This function can be used for instance to implement another function, which is strangely enough not implemented in StrUtils: a POS function that starts searching at the back of the string, working its way to the beginning.

```
Function RPos(Substr : String; S : String) : integer;

begin
  Result:=SearchBuf(S,Length(S),0,Substr,[]);
end;
```

Note that the starting position of the search is the last character of the string.

To search and replace - as in the `Replace` function presented earlier is - the SysUtils unit contains a more advanced version, which offers some extra functionality:

```
function StringReplace(const S: string;
                       const OldPattern: string;
                       const NewPattern: string;
                       Flags: TReplaceFlags): string;
```

The extra functionality is determined by the `Flags` set, which can contain the following elements:

rfReplaceAll Replace all occurrencies of `OldPattern` with `NewPattern`. By default, only the first match is replaced.

rfIgnoreCase The matches are searched ignoring case.

This function can be used in simple search/replace mechanisms in an application, as demonstrated in the following routine, which replaces one text (in an edit control named `EOld`) with another (in `ENew`) in a TMemo component (called `MText`). Additionally, there are 2 checkbox components to specify the flags (`CBIgnoreCase` and `CBReplaceAll`), and a checkbox (`CBSelectionOnly`) to select whether only the selected text in the memo should be searched:

```
procedure TMainForm.Button1Click(Sender: TObject);

Var
  rf : TReplaceFlags;
  S,N: String;

begin
  // Get text to search
  If CBSelectionOnly.Checked then
    S:=MText.SelText
  else
    S:=MText.Text;
  // Get options
  RF:=[];
  If CBreplaceAll.Checked then
    Include(RF,rfReplaceAll);
  if CBIgnoreCase.Checked then
    Include(RF,rfIgnoreCase);
  // get new text.
  N:=Stringreplace(S,EOld.Text,ENew.Text,RF);
  // Put it back in place
  If CBSelectionOnly.Checked then
    MText.SelText:=N
  else
    MText.Text:=N;
end;
```

The result can be seen in the following picture:

6 A Case Statement with strings

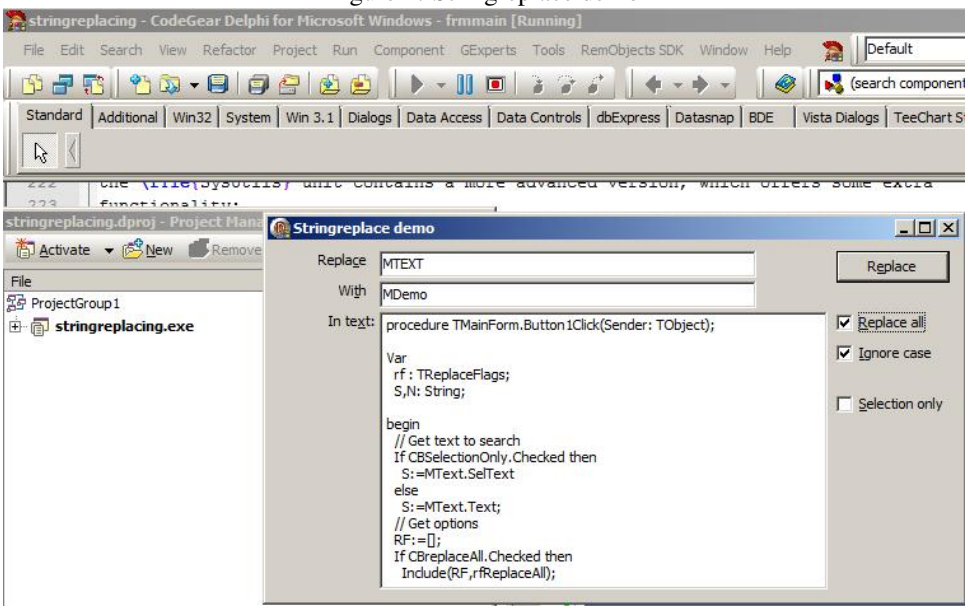
The StrUtils unit contains some more utility functions. One of these functions is an interesting one, because it alleviates a commonly heard complaint about the Object Pascal language, namely that it does not have a `Case` statement with string values. (Free Pascal has a native `Case` statement with strings as of version 2.5.1).

The function is called `IndexStr`, and its case insensitive variant is called `IndexText`:

```
// Return index of AText in AValues
function IndexStr(const AText: string;
                  const AValues: array of string): Integer;
// Return index of AText in AValues (case insensitive)
function IndexText(const AText: string;
                   const AValues: array of string): Integer;
```

They can be used to create a `Case` statement, as in this command-line examining routine:

Figure 4: Stringreplace demo




```
procedure CheckCommandLineArguments;

Const
  Cmds : Array[1..5] of string =
         ('input','output','help','version','update');

Var
  I : Integer;
  arg : string;

begin
  I:=1;
  While I<ParamCount do
    begin
Arg:=ParamStr(i);
// Strip '--' from command.
Delete(Arg,1,2);
Case IndexStr(Arg,Cmds) of
  0 : DoInput; // Zero based !
  1 : DoOutput;
  2 : ShowHelp;
  3 : ShowVersion;
  4 : DoUpdate;
else
  // Not a valid command
  Writeln('Unknown option at pos ',I,': ',Arg);
    end;
Inc(I);
end;
end;
```

A similar construction can be used in text parsers to branch execution based on a current

word.

7 conclusion

There are more string functions in the `StrUtils` unit, but most of the remaining function are simple wrappers around the functions presented here. An exception is the `SoundEx` procedure, used in 'Sounds as' functionalities, but the implementation is unfortunately useful only for the English language: results in a non-englush language such as Dutch or German are disappointing. The interested reader can consult the Delphi or Free Pascal documentation for an overview of the remaining function. The Free Pascal version of StrUtils contains a lot more functions than the Delphi version, so if compatibility is required, only the Delphi documentation should be consulted. The functions presented here do most of the hard work in text manipulation, and they should make it clear that text analysis in Object Pascal is actually quite easy, and is completely safe from buffer overruns. Nevertheless, more advanced text treatment, such as writing a scanner/parser, makes surprisingly little use of these functions. Maybe a future contribution will show why.