Strings in New Stanford Pascal

First of all: please excuse possible errors in my English; I am German and not a native English speaker … I will do the best I can.

A string in Standard Pascal usually means a packed array of char (of fixed length). I already tried to make working with arrays of char more comfortable by allowing assignments of string constants of shorter length, and recently even char array variables of shorter length; the rest of the longer target variable is filled with blanks.

And I recently allowed to use a CHAR (n) definition instead of ARRAY [1..n] of CHAR, which means the same. There is no need any more to define types for every needed length n of char arrays, because CHAR (n) is allowed everywhere where a type identifier is allowed, too. BTW: PACKED is ignored at the moment in Stanford Pascal.

But in other languages a „string“ usually means a variable which can hold a character string of varying length, for example VARCHARs or CHAR (n) VAR in PL/1.

There are different ways to implement strings; using length fields or string terminator characters (like in C). I chose length fields like in PL/1. My string implementation was inspired by the implementation of IBMs Pascal/VS, but I only had a user manual – I didn't know anything about the internal representation of the Pascal/VS strings.

After thinking some time about possible implementations, I realized that I would need two length fields for a string variable, because the Pascal/VS library supports a MAXLENGTH function and a LENGTH function – and because MAXLENGTH should work on strings passed as by-reference parameters, too, I thought that the maximum length should be recorded in the string variable. This way I came to a design, where a string variable consists of three parts:

- a maxlength field at the beginning (2 bytes)
- a length field next (2 bytes)
- and the string content; the maximum possible bytes is reserved.

That is: a variable

```
X : STRING (20) ;   // or VARCHAR (20)
```

will need 24 bytes.

Strings on the Stack

The representation mentioned before (2 length fields, MAXLENGTH and LENGTH, followed by the string content) is only used for string variables (automatic and static).

If strings are used in expressions or passed to functions or returned from functions (on the stack), another representation is used.

In this case, the strings are pushed and popped to and from the stack like other variables, and so it seemed appropriate to have a shorter representation. I decided to put only 8 bytes on the stack:

- the maxlength field (2 bytes)
- the length field (2 bytes)
- and a pointer (a cell number) to the string content

The maxlength field is always minus 1 in this case, which has the following implications:

- the content does not follow the length field, but is addressed by a pointer which follows the length field
- the maxlength is equal to the length (and has to be fetched from there)
- the „string on the stack“ cannot be expanded

In fact, strings are NEVER expanded. If strings are concatenated (for example), a new string is built on the stack which is the concatenation of the two original strings. The original strings may have been normal string variables or strings on the stack. The result is a string on the stack, using the representation described above.

There remains a problem: where will the strings be stored, that result from concatenations (or other string expressions)? In the end, the result of string expressions will be assigned to „normal“ string variables, which are (hopefully) large enough to hold the result. But the intermediate results (or results from string functions or complicated concatenations or both) have to be stored anywhere; same goes for string expressions involved in string comparisons. They may need up to (several times) 32 k bytes temporarily.

To handle this, I defined a new storage area, called the „string workarea“. Its size can be determined at startup time, and it is used by the string related P-Codes to put the temporary strings there.

The String Workarea

The string workarea is used by certain P-Codes to store temporary strings.

For example:

if two strings, say A and B, and a char constant, say 'X', are concatenated and assigned to a string variable C in the end, the following happens:

first, the string variable A is pushed to the stack, that is, an 8 byte representation of A, consisting of its actual length and a pointer to the content, is pushed to the stack. Then the same is done for B. There is no need to copy the content of A or B to the string workarea, because the following concatenation will do it, anyway. The concatenation P-Code concatenates the contents of A and B, this time copying the result to the string workarea. The „strings on stack“ representing A and B are popped from the stack, and a new element, describing the result, is pushed. The „used pointer“ of the string workarea is incremented.

Now, the character X is converted to a string of length 1; that is, another string representation is pushed to the stack. The next concatenation P-Code does again the concatenation. The stack now contains one string element, describing the concatenation of A, B and the character 'X'. In the string area, we have three strings, but two of them are obsolete already: the concatenation of A and B (obsolete), the one-byte string containing the 'X' (obsolete) and the final result.

Last step: the final result is stored to the normal string variable C. The length of the string on the stack is checked against the allowable maxlength of C; if it is larger, a runtime error occurs.

To prevent memory leaks in the string workarea, the „used pointer“ now is reset to the value that it had at the beginning of the statement. There are certain P-Codes that save and restore the string workarea „used pointer“, and these P-Codes are inserted into the P-Code stream, whenever necessary. In every function, certain locations (memory cells) are reserved, where these string workarea „used pointers“ can be stored.

BTW:

at the time of this writing, the new string P-Codes (all beginning with the letter V), are not yet available on the mainframe implementation. That means: Pascal strings can only be used by the non-mainframe versions of Stanford Pascal, at the moment.

Operations on Strings

First of all, strings can be assigned to other strings, regardless of their defined maxlength. Only the actual length is checked, and there will be a runtime error, if the target string variable is not large enough to hold the result.

Strings can also be assigned to ARRAYs of CHAR, as long as the char array is large enough to hold the string (depending on the string's actual length, not the maxlength). Again, there will be a runtime error, if the source string is too large.

String constants are in fact „char array constants“, but they can be assigned to strings directly without problems (as long as the string variable is large enough to hold the string constant).

On the contrary, char array variables and single chars cannot be assigned directly to string variables. But (similar to Pascal/VS) there is a standard function STR which allows such conversions.

Examples:

```
var C1 : CHAR ;
    C20 : CHAR ( 20 ) ;
    VC20 : STRING ( 20 ) ;

    // some possible assignments

    VC20 := 'A' ;              // VC20 contains A and blanks
    VC20 := 'Test Varchar' ;   // assignment of string constant
    VC20 := STR ( C20 ) ;      // STR function needed
    C20 := VC20 ;              // direct assignment possible
```

There is no problem, if you put an (unneeded) STR call around the char and string constants, too.

BTW, the rules for assigments between strings and arrays of char are much the same as in Pascal/VS … the STR function was borrowed from Pascal/VS, too.

I forgot to mention: Strings are limited to 32767 Bytes; string constants (at the moment) may not exceed 254 bytes.

Concatenation of Strings

Strings can be concatenated using the || operator.

Example:

```
C20 := 'Bernd ' || 'Oppolzer' ;
WRITELN ( 'C20: ' , C20 ) ;
VC20 := 'Bernd' ;
VC200 := VC20 || VC20 ;
WRITE ( 'VC200: ' ) ;
WRITELN ( '<' , VC200 , '>' ) ;
VC200 := VC20 || ' Oppolzer' ;
VC200 := VC20 || ' Oppolzer' || ' Leinfelden' ;
VC200 := VC20 || ( ' Oppolzer' || ' Leinfelden' ) ;
VC200 := ( VC20 || ' Oppolzer' ) || ' Leinfelden' ;
VC200 := ( ( VC20 || ' Oppolzer' ) || ' Leinfelden' ) ;
VC200 := VC20 || STR ( ' Oppolzer' ) ||
         STR ( ' Leinfelden' ) ;
VC200 := VC20 || ' ' || VC20 ;
VC200 := VC20 || ' dazwischen ' || VC20 ;
VC200 := VC20 || ' dazwischen ' || STR ( C20 ) ;
VC200 := STR ( C20 ) || ' dazwischen ' || VC20 ;
```

this coding has been used to test several variants of concatenation during the development phase.

The || operator is handled by the compiler at the same level as the binary plus and minus operators for arithmetic.

Accessing Parts of Strings

You can access single characters of strings simply by using the normal array notation; that is: the first character in the string has the index 1, the second has index 2 and so on. The current length of a string can be examined by using the LENGTH function, and the maximum allowable length of a string by using the MAXLENGTH function. MAXLENGTH will also work on strings passed as var parameters to a procedure or function and will yield different results, depending on the definition ot the string variable passed as parameter (see „conformant string parameters“ later).

Another possibility to access parts of a string is using the SUBSTR function; see also later.

Example program:

```
program TESTSTR1 ( OUTPUT ) ;

var S : STRING ( 20 ) ;
    I : INTEGER ;

begin (* HAUPTPROGRAMM *)
  S := 'Bernd' ;
  WRITELN ( 'maxlength = ' , MAXLENGTH ( S ) ) ;
  WRITELN ( 'length    = ' , LENGTH ( S ) ) ;
  for I := 1 to LENGTH ( S ) do
    WRITE ( S [ I ] , ' ' ) ;
  WRITELN ;
end (* HAUPTPROGRAMM *) .
```

this program prints:

```
maxlength =           20
length    =            5
B e r n d
```

Passing Strings as Parameters to Procedures and Functions

This sample program shows three different parameter passing mechanisms available for strings: by value, by reference (var) and by reference using dummy arguments (const).

```
program TESTSTR2 ( OUTPUT ) ;

var S : STRING ( 20 ) ;

procedure TESTVALUE ( S : STRING ( 30 ) ) ;

   begin (* TESTVALUE *)
     WRITELN ( 'testvalue: maxlength = ' , MAXLENGTH ( S ) ) ;
     WRITELN ( 'testvalue: length    = ' , LENGTH ( S ) ) ;
     WRITELN ( 'testvalue: content   = ' , S ) ;
   end (* TESTVALUE *) ;

procedure TESTCONST ( const S : STRING ) ;

   begin (* TESTCONST *)
     WRITELN ( 'testconst: maxlength = ' , MAXLENGTH ( S ) ) ;
     WRITELN ( 'testconst: length    = ' , LENGTH ( S ) ) ;
     WRITELN ( 'testconst: content   = ' , S ) ;
   end (* TESTCONST *) ;

procedure TESTVAR ( var S : STRING ) ;

   begin (* TESTVAR *)
     WRITELN ( 'testvar: maxlength = ' , MAXLENGTH ( S ) ) ;
     WRITELN ( 'testvar: length    = ' , LENGTH ( S ) ) ;
     WRITELN ( 'testvar: content   = ' , S ) ;
     S := 'Hugo' ;
   end (* TESTVAR *) ;

begin (* HAUPTPROGRAMM *)
  S := 'Bernd' ;
  TESTVALUE ( S ) ;
  TESTVALUE ( 'Bernd ' || 'Oppolzer' ) ;
  TESTCONST ( S ) ;
  TESTCONST ( 'Bernd ' || 'Oppolzer' ) ;
  TESTVAR ( S ) ;
  WRITELN ( 'main: S after testvar = ' , S ) ;
end (* HAUPTPROGRAMM *) .
```

Passing Strings „by value“

When a string is passed by value, it is **copied** into the value parameter, which is a **local variable** of the called procedure or function. This means that the parameter has its own attributes which may be different from the attributes of the passed parameter.

In the example above, the procedure TESTVALUE gets a STRING (30) parameter. BTW: you may notice, that a type identifier **with parameters** is allowed in function definitions (standard Pascal allows only identifiers at this place, no parameters).

When calling TESTVALUE, strings with other attributes may be passed as parameters. See in the sample program:

```
  TESTVALUE ( S ) ;
  TESTVALUE ( 'Bernd ' || 'Oppolzer' ) ;
```

The routine TESTVALUE prints the LENGTH and the MAXLENGTH of the parameter. Because the parameter is a local variable of TESTVALUE and has its own attributes, the program consequently prints:

```
testvalue: maxlength =          30
testvalue: length    =           5
testvalue: content   = Bernd
testvalue: maxlength =          30
testvalue: length    =          14
testvalue: content   = Bernd Oppolzer
```

that is: the MAXLENGTH is always 30. Inside the TESTVALUE procedure, new values could be assigned to the parameter S, but these changes will **not be visible** outside of TESTVALUE.

Passing Strings „by reference“ (var parameters)

When a string is passed by reference, **its address** is passed to the called procedure or function. **Only string variables** can be passed, no expressions (hence the term „var parameter“). Every reference to the parameter inside the procedure is done using the passed variable address, and because the address includes the MAXLENGTH and the LENGTH fields of the string variable, the MAXLENGTH and LENGTH attributes are known inside the called procedure, too.

If a new value is assigned to the string inside the called procedure, this has the following implications:

- the new value will be checked against the MAXLENGTH of the target variable (if it is too large, a runtime error will occur)
- the new value will be visible outside of the called procedure (of course).

In the sample program, I called TESTVAR and passed the variable S from the main program. The procedure TESTVAR prints the MAXLENGTH and the LENGTH of the passed parameter. I then assigned a new value to the var paremeter inside the procedure. The output looks like this:

```
testvar: maxlength =           20
testvar: length    =            5
testvar: content   = Bernd
main: S after testvar = Hugo
```

What is interesting about the TESTVAR definition:

the procedure uses a **conformant string parameter**, that is: a parameter with the length omitted. This is in fact the only possible way to specify var parameters for strings; the length will always be inherited from the passed argument.

```
procedure TESTVAR ( var S : STRING ) ;

...
```

Passing Strings using dummy arguments (const parameters)

Pascal VS introduced another parameter passing mechanism known as **const parameters**. In this case, an address is passed to the procedure or function, much like with var parameters. But in contrast to var parameters, **expressions** can be specified on const parameters, too. If necessary, the compiler builds a **temporary variable** where it stores the result of the expression and passes the address of this temporary variable (a so called dummy variable) to the subroutine. Even if the subroutine would change the passed parameter, this would have no effect on the passed parameter.

I decided to implement the const parameters in this compiler release, too. This makes it possible to implement several string related functions like SUBSTR, INDEX, VERIFY, TRANSLATE and so on in Pascal (see later). Const parameters are also implemented for every other type, but they still need some more testing. And: at the moment the compiler does not check that there are no assignments to const parameters inside the called procedure or function. With strings at least, assignments inside a function or procedure will not work, because const parameters have a MAXLENGTH attribute of minus 1. So this way they are protected from assigments at the moment.

See the example program above; const parameters are defined much the same way like var parameters, but even complicated string expressions can be passed to them. The example program prints this from the procedure testconst:

```
testconst: maxlength =          -1
testconst: length    =           5
testconst: content   = Bernd
testconst: maxlength =          -1
testconst: length    =          14
testconst: content   = Bernd Oppolzer
```

Const string parameters should normally be **conformant string parameters** (without length specified); the length will be inherited from the passed argument.

```
procedure TESTCONST ( var S : STRING ) ;

...
```

Functions returning Strings

A Pascal function may **return a string result**. The result must be a **conformant string** (no length specified). In fact, the string length is determined by the string function in the moment when it assigns a string to the string function name.

Simple example:

```
function STRFUNC ( const X : STRING ; const Y : STRING ) : STRING ;

   begin (* STRFUNC *)
     STRFUNC := X || '/' || Y ;
   end (* STRFUNC *) ;
```

This function gets two string parameters; it does a concatenation of the two strings with a slash in between and returns this as function result.

When I started to implement functions like SUBSTR etc. in Pascal, I soon observed that I need some additional features to be able to better control the result of string functions. So I introduced some builtin **helper functions** that are compiled inline (no real function call) and allow for better control of function results and string handling. Some of these functions are kind of dangerous and must be handled with care.

The following list contains ALL builtin functions added during Pascal string development, which are compiled inline and don‘t really call a function or procedure.

STR	this function converts char arrays and single chars to strings
MAXLENGTH	gets the maximum length of a string variable
LENGTH	gets the current length of a string variable
STRRESULT	is a function of type STRING which retrieves the current function result, when inside a string function
STRRESULTP	is a function of type pointer to CHAR, which points to the content of the current function result, when inside a string function
REPEATSTR	REPEATSTR (s, n) returns the string s repeated n times. This function is implemented using a new P-Code instruction, and it is used inside string functions to build new strings, containing of n blanks (for example)
RESULTP	is a function returning pointer (ANYPTR) to the function result (usable in every function)

Implementing SUBSTR in Pascal

Using the helper functions mentioned above, it was now possible to implement a function like SUBSTR completely in Pascal. In fact, this is how it is done at the time of this writing; the SUBSTR function is known implicitly by the compiler, and it is translated to a library call, which means a call to a function which resides in the PASLIBX module. This function (called $PASSTR1) has an additional function code as first parameter and implements not only SUBSTR, but also DELETE, RTRIM, LTRIM, TRIM and COMPRESS (see later).

I'll show you below a development version of SUBSTR, which is not very different from the version of PASLIBX. Of course, it would be better to compile SUBSTR inline by means of a new P-Code instruction. This way, SUBSTR could generate still more efficient code, for example when the arguments to SUBSTR are integer constants. Maybe I will do this later, but at the moment I wanted to do a fast proof of concept, and using the library call mechanism and implementing all the functions in Pascal, it was possible to implement (almost) the whole Pascal/VS string library in roughly three days.

This is what SUBSTR2 (development version of SUBSTR) looks like:

```
function SUBSTR2 ( const SOURCE : STRING ; START : INTEGER ; LEN :
                 INTEGER ) : STRING ;

   var X : INTEGER ; P : ANYPTR ; Q : ANYPTR ;

   begin (* SUBSTR2 *)
     if LEN < 0 then
       begin
         if START > LENGTH ( SOURCE ) then
           EXIT ( 1201 ) ;
         LEN := LENGTH ( SOURCE ) - START + 1 ;
       end (* then *)
     else
       begin
         X := START + LEN - 1 ;
         if X > LENGTH ( SOURCE ) then
           EXIT ( 1201 ) ;
       end (* else *) ;
     SUBSTR2 := REPEATSTR ( ' ' , LEN ) ;
     P := STRRESULTP ;
     Q := ADDR ( SOURCE [ START ] ) ;
     MEMCPY ( P , Q , LEN ) ;
   end (* SUBSTR2 *) ;
```

The SUBSTR2 function from the previous page needs some additional comments:

a) SUBSTR may be called with two or three parameters; if two, the compiler inserts a minus one as the third parameter (which means, SUBSTR from the given position to the end of the string). This is how SUBSTR2 works, too.

b) you may notice the use of the helper function REPEATSTR and STRRESULTP, which made the implementation of SUBSTR2 (and SUBSTR) easy

c) REPEATSTR and MEMCPY are implemented by inline calls (fast) – so, apart from the overhead to call the function, SUBSTR2 (and SUBSTR) should be reasonably fast. The function result is built in the string workarea (see earlier paragraph).

d) EXIT (1201) simply terminates the Pascal program with a „stop code“ of 1201. Maybe in a later version it would be better to throw a more significant run time error like STRINGRANGE. But this has to be done in a consistent manner in the different runtimes (Mainframe: Pascal monitor PASMONN; others: in the P-Code interpreter PCINT), which needs some work to be done.

More Functions from the Pascal/VS Library (and others)

After I implemented SUBSTR successfully in PASLIBX (see above), I continued to implement most of the other functions of the Pascal/VS library … and some more. All these functions are known to the compiler (no definition needed) and work in the same way as described in the Pascal/VS documentation. I also repeat here the helper functions mentioned earlier.

Summary of all string related functions (at time of this writing):

STR	this function converts char arrays and single chars to strings
MAXLENGTH	gets the maximum length of a string variable
LENGTH	gets the current length of a string variable
STRRESULT	is a function of type STRING which retrieves the current function result, when inside a string function
STRRESULTP	is a function of type pointer to CHAR, which points to the content of the current function result, when inside a string function
REPEATSTR	REPEATSTR (s, n) returns the string s repeated n times. This function is implemented using a new P-Code instruction, and it is used inside string functions to build new strings, containing of n blanks (for example)
RESULTP	is a function returning pointer (ANYPTR) to the function result (usable in every function)
SUBSTR	SUBSTR (s, n, m) or SUBSTR (s, n) – gets the Substring from s starting at position n with length m (first format) or the rest of the string (2nd format); same as in Pascal/VS and PL/1
DELETE	deletes portions of a string; the parameters are the same as with SUBSTR. DELETE works the same as its Pascal/VS counterpart
RTRIM	trims spaces on the right (like TRIM in Pascal/VS)
LTRIM	trims spaces on the left (like in Pascal/VS)
TRIM	trims spaces on both sides (different from TRIM in Pascal/VS !!)
COMPRESS	replaces sequences of spaces in the string by only one space (like in Pascal/VS)
INDEX	INDEX (s1, s2) returns the index of the first occurence of s2 in s1; zero if not found. Same as in Pascal/VS and PL/1.
VERIFY	VERIFY (s1, s2) returns the index of the first character in s1 which is not in s2 (same as in PL/1)
TRANSLATE	TRANSLATE (s1, to, from) translates s1 controlled by one or two translate tables (see PL/1)

I hope you like the new string features of New Stanford Pascal;
please send comments and suggestions to

berndoppolzer@yahoo.com

or

bernd.oppolzer@t-online.de