A Better Way to Combine Efficient String Length Encoding and Zero-termination

C. Bron
E.J. Dijkstra
Department of Mathematics and Computing Science
University of Groningen
Netherlands

December 13, 1988

Abstract

In this note we describe conventions for string handling, and in particular for efficient string length encoding. These conventions are based on the (C-language) zero-terminator convention, and assume that the size of the area containing the string is known. They do not require any special provisions on the part of the language implementation. The mechanism to be described caters smoothly for strings of arbitrary length. The notes are based on several years of experience within the framework of Modular Pascal.

1 Introduction

In [1], "Some Sad Remarks About String Handling in C", Paul Abrahams comments on the inherent inefficiencies of the C programming language convention to terminate strings with an (ASCII-)null character. The inefficiencies arise from determining the length of the string, which necessarily takes time proportional to the length of the string, presuming the string can be accessed from one end only.

Abrahams proposes to encode the length of the string explicitly, as the (pre-)first byte of the string, thus providing increased efficiency in operations like `strcat` which need the length of the first string in order to append the second.

Since the convention to add an additional length byte at the front of the string has already found its way into several language implementations, notably Turbo Pascal[1] [2], and may be adopted in the course of several ongoing language standardization efforts, it seemed worthwhile to publicize an attractive alternative which we have adopted in Modular Pascal [3], a home grown extension of Pascal, with which we have gained positive experiences in recent years.

The main advantages of our approach are that it provides the efficiency gain of Abrahams' proposal without any impact on the existing language implementation, and requires very little discipline of the user. The enforcement of this discipline is supported by a library module providing a limited but very widely applicable set of string handling routines.

[1]Turbo Pascal is a Registered Trade Mark of Borland International

Before proceeding with the details, we present the essence of our solution: the length of the string is not encoded at the front of the string but *at the end*, and only if there is sufficient unused space at the end. The only requirement on the language implementation (which may not be fulfilled by C) is that the size of the string-*container* be known, either at run time or at compile time.

2 String Containers

In Modular Pascal we may have to deal with three kinds of strings

1. string literals
2. declared string variables of type `PACKED ARRAY [....] OF char`
3. strings passed to procedures as parameters (cf. the conformant array parameters of ISO-Pascal)

For each of these string kinds the size of the container is known:

1. it equals the number of characters in the string denotation (compile time constant)
2. it follows from the bounds in the array type specification (compile time constant)
3. it is an additional data item passed to a procedure as part of the string parameter (run time constant). Note that according to the ISO-Pascal [4] standard both lower and upper bound must be passed as part of the conformant array parameter since they can be addressed as named constants.

In Modular Pascal, the standard functions `lwb(..)` and `upb(..)` yield lower, c.q. upper bound of the index range for cases 2) and 3). In what is to follow, we make two simplifying assumptions, viz. that for every string container `s: lwb(s) = 0` and that `upb(s) >= 0`, in other words we do not consider containers that can not contain at least one character. Neither assumption is essential but they both contribute to the simplification of an algorithm for finding the length of a string. The first simplification is in some cases fully language enforced (as in C) and in others only partially (as in Modula-2 [5]) where a shift of index range takes place when a character array is passed to a procedure as an open array parameter. The second assumption is fulfilled by the Pascal Standard which forbids the empty string literal.

3 String Length Encoding

In the Modular Pascal approach we combined the following principles:

- no deviation from language definition
- no hidden, additional data fields associated with strings
- no compiler action required
- null-termination wherever possible
- fast string length encoding

For the sake of the description we will consider the component type of strings as (small) integers in the range 0..255 instead of as characters, therefore we will not make explicit use of the Pascal type transfer ord.

A vital role in encoding the length of a string s is played by the last element of s: s[upb(s)]. Its value will primarily be interpreted as the distance from the end of s where the null-terminator may be found if the string s contains a null-terminator at all. This encoding has a number of attractive properties:

- if the last character in the string container is a null-character it indeed locates the position of the (terminating) null-character.
- if the position in the string so designated does not contain a null-character, the string is 'full', i.e. the number of relevant characters in the string equals the size of the string container, and it is, consequently, not null-terminated.
- if string containers are long (longer than, say, 256 bytes) and almost full, only a small (byte-sized) integer will do to encode the end-of-string position. If the long string container is far from full, the tail of the string container has plenty of unused space to encode by means of some 'escape value' the position of the terminating null-character as a more than byte-sized integer.

These aspects have been incorporated in the function eos(s), which – for strings indexed from 0 – is the same as length(s). Another interpretation of eos(s) is as the position where the terminating null-character *may* be found. For strings not indexed starting from 0 we obviously have the equivalence:
length(s) = eos(s) - lwb(s).
A non-null terminated string can be recognized by:
eos(s) = upb(s) + 1
The implementation of the eos function follows directly from the above:

```
CONST esc_val = 255 (*?*);
FUNCTION eos(CONST s: STRING): integer;
  VAR pos: integer;
BEGIN pos := upb(s) - s[upb(s)];
      eos := upb(s) + 1; (* default for non null-terminated *)
      IF pos >= 0 THEN
         IF s[pos] = null_char
         THEN eos := pos ELSE
         IF s[upb(s)] = esc_val
         THEN BEGIN
              pos := s[upb(s) - 2] * 256 + s[upb(s) - 1];
              (* in this case the encoding is relative to location 0 *)
              IF (pos >= 0) AND (pos <= upb(s) AND (s[pos] = null_char)
              THEN eos := pos
         END
END (* eos *);
```

Note that in spite of the apparent complexity of the above implementation of eos, the common cases, especially the 'short', null-terminated string, are dealt with very efficiently. The above may be improved on by moving the default assignment of the function result to the individual

default branches of the conditional statements. It was only written this way for clarity of presentation. In the last (pos >= 0 .....test the term (pos >= 0) may be omitted in the case of 32-bit integers, for the result cannot be negative. If it can be guaranteed that strings do not contain the character value used for esc_val then the only reason for this value to appear in s[upb(s)] is that it was deliberately put there for the encoding of the null-terminator position. This requirement seems reasonable and reduces the algorithm to:

```
...IF s[upb(s)] = esc_val
   THEN eos := s[upb(s) - 2] * 256 + s[upb(s) - 1]
```

In the case that a two byte integer must be used to encode the distance of the end-of-string to the end-of-container, the routine responsible for encoding the string length (terminate must ensure that s[upb(s) - esc_val] <> null_char otherwise this position would erroneously be recognized as the end-of-string position.

4 The User's Discipline

Little discipline is required in this respect, because:

- Any string literal not containing a null-character satisfies the requirements.
- Any string literal only containing a null-character in its last position satisfies the requirements.
- Any string produced as a result of an operation from the STRINGS module satisfies the requirements.
- Any string produced by any other module from the Modular Pascal library satisfies the requirements because the library obeys the preceding three rules.
- Any string smaller than its container contains a terminating null-character.
- Where terminating null-characters are required, they can either be supplied at the end of *string literals*, or – in cases where their presence is not certain – they are obtained by copying the string into a container of sufficient size, using the procedure assign from the STRINGS module.
- When the characters of a string have been filled in without the help of procedures from the STRINGS module, the string can be made to satisfy the requirements by calling for that string: terminate(s, e) where e is the position that will subsequently be the result of eos(s). The only requirement for terminate is that s[0..e-1] contain meaningful (non-null) characters.

[Remark]
The requirement that a string not contain null-characters other than in the terminating position can be relaxed if it can be ascertained that operations on the string do not proceed 'from left to right' until a null-character is encountered, but make use of the result of eos(s) instead. Such use of a string is suggested by the example in [1, page 66]. Obviously, the proposed organization of strings does not prohibit in any way such an application.
[End of Remark]

The last part of the discipline imposed on the user concerns strings that *are* null-terminated but originate from 'somewhere'. Such a string may be adapted to the proposed conventions by means of `accept(s)`. The role of `accept` is (of course) to find the null-character by a left-to-right scan of `s` and subsequently to encode its length in the described way.

5 A truly different alternative

Another, radically different, way to encode the length of a string is again based on the possibility of locating both the beginning and the end of the string container. It further precludes the use of strings that contain interior null-characters.

We postulate that, within the string container, *all* characters following the last significant character should be null-characters. It is then possible to locate the end-of-string position by a binary search (!!) because the contents of the string container monotonically 'increase' from non-null to null-characters!

According to this convention string literals, either with or without null-terminator are acceptable. String variables must, however, be set to all null-characters when they are created or initialized. After that, strings continue to satisfy the conventions as long as they are not shrunk. This can only take place on account of `terminate(s,e)`, therefore null-characters should now be added upto the first position `p >= e` for which `s[p] = null_char`. It is the user's obligation to guarantee that `s[0..e-1]` not contain nulls.

We have no experience with an actual implementation of this alternative, but it is almost certainly an improvement over the linear length search described in [1] for C. Furthermore the proposed implementation is very elegant since no distinction whatsoever needs to be made between short and long string containers.

6 A STRINGS Library Module

Textbooks on present day programming languages abound with descriptions of procedures (either library defined or standard) for string handling, so why bother to present 'just another interface'. The motivation is that the present STRINGS module of Modular Pascal is not the result of a quick design, but evolved over a number of years of careful deliberation into its present form. As a result of this evolution it has not only proven to be widely applicable, but it also contains several novel features not found in other string handling packages.

For a better understanding of the given procedure headings, note that apart from the Pascal mechanisms of value- and var-parameters Modular Pascal provides const-parameters, which may be thought of as 'write-protected' var-parameters, i.e. no value can be assigned to them, nor can they be passed to procedures in var-parameter positions. The symbol `STRING` stands for a conformant packed character array parameter.

So here we go. The first set can be considered elementary and follows immediately from the above descriptions:

```
CONST null_char = chr(0);

PROCEDURE init( VAR s: STRING );
(* makes  s an empty string *)

FUNCTION is_empty( CONST s: STRING ): boolean;
```

This function is actually superfluous because its result is equivalent to that of:
`eos(s) = lwb(s)`
or, alternatively, to that of
`s[lwb(s)] = null_char`
for all but the empty string container.

```
PROCEDURE terminate( VAR s: STRING; e: integer );

PROCEDURE accept( VAR s: STRING );

PROCEDURE assign( VAR sl: STRING; CONST sr: STRING );

PROCEDURE append( VAR sl: STRING; CONST sr: STRING );
```

This is, as can be expected, the equivalent of C's `strcat`.

```
PROCEDURE append_char( VAR s: STRING; c: char );
```

So far, there is very little new under the sun, but the following set of procedures illustrates a very useful feature: an additional position (integer) variable is passed to the string handling procedure by means of which the procedure can communicate its progress to the calling environment. In all cases where two strings are involved, the first one contains the result and is therefore terminated, so the string position designates a position in the second string.

```
FUNCTION next_char( CONST s: STRING; VAR pos: integer ): char;
```

`pos` must be initialized to the position of the first character of `s` to be delivered, and is updated to the next. Calling this function for every single character to be retrieved is not the utmost of efficiency, but the elegance of this function lies in its delivering a null-character after the last significant character, regardless of whether the null-character is actually encoded in the string.

```
PROCEDURE append_upto( VAR sl: STRING; CONST sr: STRING;
                       VAR pos: integer; c: char
                     );
```

This procedure appends the part of `sr` onto `sl`, starting at `sr[pos]` upto (but not including) the first occurrence of the character `c` in `sr`, or (if `c` is not found) upto the end of `sr`. Upon completion `pos` satisfies:
`(pos = eos(sr)) OR (sr[pos] = c)`
The following procedures are analogous, except for their termination condition:

```
PROCEDURE append_upto_ccond( VAR sl: STRING; CONST sr: STRING;
                             VAR pos: integer;
                             FUNCTION ccond( c: char ): boolean
                           );
```

Upon completion: `(pos = eos(sr) OR ccond(sr[pos]))`
And to allow other forms of termination than on account of the character value itself:

```
PROCEDURE append_upto_pcond( VAR s1: STRING; CONST sr: STRING;
                             VAR pos: integer;
                             FUNCTION pcond( p: integer ): boolean
                           );
```

Upon completion: (pos = eos(sr) OR pcond(pos))
Note the interesting hierarchy present in this set of procedures; in the following list each one can be expressed in terms of the next one!

```
assign
  \/    (* after an init *)
append
  \/    (* using null_char for c *)
append_upto
  \/    (* using sr[pos] = c for ccond(c) *)
append_upto_ccond
  \/    (* using ccond(s[p]) for pcond(p) *)
append_upto_pcond
```

Finally, string comparison follows the suggestion made in [1, page 66] where an integer result is returned, of which only its comparison with 0 is relevant. This is a very general technique that can be applied to virtually any data type for which a total ordering is defined. It not only obviates the necessity to write several comparison operators (we need at least two: $<$ and $=$, the others can be made by negation and interchanging of operands), but it also may significantly save in cases where the outcome of a single boolean operator is insufficient, as, for example, in dealing with binary search trees. (Are we reinventing FORTRAN's three way branch?)

```
FUNCTION compare( CONST s1, s2: STRING ): integer;
VAR p1, p2: integer; c1, c2: char;
BEGIN p1 := 0; p2 := 0;
      REPEAT c1 := next_char( s1, p1 ); c2 := next_char( s2, p2 )
      UNTIL (c1 <> c2) OR (c1 = null_char);
      compare := ord(c1) - ord(c2)
END;
```

Note how even a small algorithm like this can be full of subtlety.

6.1 Comparison with a recent proposal

The use of an additional position parameter in many of the string copying procedures compares favorably with a very recent proposal[6], especially where efficiency is concerned.

The first example shows the construction of a new file name from a given one, merely by changing its suffix (extension).

```
pos := 0;
append_upto( NewFileName, OldFileName, pos, '.');
append( NewFileName, '.PAG')
```

Note that this does the trick, regardless if the old file name did or did not have a suffix. In the example the position parameter is not used, but it could have been used to extract

the suffix of the old file name. Needless to say that this operation is itself encapsulated in a procedure `set_suffix` which can be found in a Modular Pascal library procedure for file name manipulation.

The second example concerns a

```
FUNCTION Parse( VAR source: string; separator: string): string;
```

which splits the given string `source` in a header part (preceding the separator), which is delivered as the function result, and the tail part (following the separator) which is left in `source`. The implementation of `Parse` as given in [6] is itself inefficient because the separator string must be located twice. This can easily be remedied by resorting to the 'low-level' function `pos`. Worse is the inefficiency in the example demonstrating the use of `Parse`, where a command(string) with a first separator ':' and further separators ',' is split into its constituents. The tail of the input string is copied onto itself as many times as the number of constituents in the command:

```
commandname := Before( command, ':' );
parms := After( command, ':' );
i := 0;
WHILE Length( parms ) > 0 DO BEGIN
      i := i + 1; parm[i] := Parse( parms, ',' )
END
```

With the position parameter of our STRINGS module, the solution (using a small local declaration for text compactification) would read:

```
PROCEDURE ass_upto( VAR part: STRING; sep: char );
BEGIN init( part );
      append_upto( part, command, pos, sep );
      pos := pos + 1
END;
.....
i := 0; pos := 0; epos := eos( command );
ass_upto( commandname, ':' );
WHILE pos < epos DO BEGIN
      i := i + 1; ass_upto( parm[i], ',' )
END
```

7 Concluding Remarks

We have shown that it is possible to combine zero-termination (as is required by some system environments) and an efficient determination of string length without making any demands on existing language implementations. The proposal will work for any Pascal-like implementation of strings. In particular, it should be noted that it can be adopted for Standard Pascal (without conformant array parameters) by defining a type `string` of sufficient length, and adding, for a sufficient number of commonly expected lengths, routines of the form:
`assign_10( VAR sl: string; sr: string_10 )`
the function of which typically consists of copying `sr` into `sl`, stripping off trailing blanks, and terminating `sl` in the proper manner. Most Pascal implementations do not need such a 'kludge', as they accept short string literals where longer ones are expected.

References

[1] Paul W. Abrahams, *Some Sad Remarks About String Handling in C.*, SIGPLAN Notices Vol.23, #10 (Oct. 88), 61-68

[2] Turbo Pascal 4.0 Owner's Handbook, Borland International, Scotts Valley, CA, U.S.A.

[3] C. Bron, E.J. Dijkstra, Report on the Programming Language Modular Pascal (3d Ed.), Department of Computing Science, University of Groningen

[4] K. Jensen, N. Wirth, A.B. Mickel, J.F. Miner, Pascal User Manual and Report (3d Ed.), Springer Verlag 1984

[5] N. Wirth, Programming in Modula-2, Springer Verlag 1982

[6] Dick Pountain, Untangling Pascal Strings, BYTE (Dec. 1988), 307-314