Character string handling in FORTRAN

R. A. Reynolds

C/o Cusdin, Burden & Howitt, Chartered Architects, 1-4 Yarmouth Place, Piccadilly, London W1Y 8JQ

This paper describes a set of subprograms, to be called from a FORTRAN program, that will carry out various operations on variable length strings. A justification for such a facility together with a brief description of similar facilities implemented or proposed in FORTRAN is given. Some conclusions drawn after implementation and use of the system on various machines are briefly given. The desirability of adding variable length strings to the ANSI FORTRAN standard is discussed.

(Received May 1976)

It was recognised quite early in the development of computers that there is a need to work with strings of characters, as well as with numbers. Although FORTRAN did not provide facilities for handling these, another early high level language, COBOL, did. COBOL incorporates fixed length character strings defined as variables of class ALPHANUMERIC. The usual operations of assignment, comparison, concatenation and so on are allowed.

The situation has been changing rather slowly, but is in favour of incorporating character handling in high level languages. FORTRAN and ALGOL 60 did not originally allow for strings, but ALGOL 68 allows for both fixed and variable length strings, and the proposed new American National Standard FORTRAN (1976) incorporates variables of type CHARACTER which are fixed length strings. In fact almost all the recently introduced languages, both at an elementary level, such as BASIC, and at an advanced level, such as PL/I, incorporate character string variables.

The situation with regard to FORTRAN has not been entirely resolved because the new standard only allows for fixed length strings. This paper discusses the need for variable length strings and the advantages and disadvantages of using them. It also describes a set of routines that have been written to handle such strings.

1. The current situation

The only official concession that the FORTRAN standard (1966) has made to character handling is in the provision of the 'A' ( alphanumeric ) format specification. The A-format allows a string of characters to be input or output from a variable; typically an integer array will be used. Once stored, the characters must be treated as ordinary numbers, which makes many operations difficult, although Day (1972) has described some useful techniques to facilitate this. Comparison, for instance, can be awkward because the sign bit may be modified. If the characters are packed several to a word the coding becomes machine dependent because of the variations between machines in the number of characters to the full word. Dissection or concatenation of packed strings is also an awkward matter, involving complicated shifting and masking which may have to be done in assembly language. On the other hand, holding the strings at one character per word will simplify the coding at the expense of wasting typically three-quarters of the storage space.

The situation has been eased to some extent by the introduction of the ENCODE and DECODE instruction into many FORTRAN compilers. These instructions allow the transmission of data from one set of variables to another under format control; thus the packing and unpacking of character strings is greatly simplified. It is also possible to convert numbers into character form and vice versa.

The drawbacks are that the coding is still machine dependent in respect of the number of characters per word, and in that the ENCODE/DECODE syntax has not been standardised and varies from compiler to compiler. The instructions are also rather clumsy and inflexible and, in the author's opinion, difficult to understand. The new standard incorporates the facilities of ENCODE/DECODE, via the READ/WRITE statements, in a rather more machine independent form.

A few FORTRAN compilers have embedded true string manipulation into the language; a popular example being Honeywell's FORTRAN Y (GEC, 1972) implemented on the Series 6000 computers. FORTRAN Y allows for fixed length strings. Assignment and comparison can be done in the usual FORTRAN syntax: blank fill or truncation being used as necessary where the strings involved are of differing lengths. Other facilities such as concatenation and dissection are provided through the use of built-in functions.

The compilers on time sharing systems tend to provide more advanced facilities than the equivalent for batch systems. One such compiler is the FIV FORTRAN (GEC, 1973) available on the Honeywell Mark III service. This compiler allows for variable length strings and also provides a fairly complete range of functions to manipulate them. The characters must be input and output through arrays, routines being provided for their transformation into string variables.

Hanson (1974) has described a system that uses complex variables to delimit character strings held in a separate storage array. The technique is useful and reasonably machine independent, but does not provide for 'garbage collection', that is, the releasing of the storage space allocated to a string when that string is no longer referred to.

Hertweck (1970) has described a comprehensive variable length string manipulation package, using real variables to reference strings held in a storage array. Storage allocation is based on list processing techniques. The package is written in IBM 360/370 assembler and does not claim to be transportable.

A number of other examples might be given, but it is clear that there must be some considerable pressure for string handling. The reason for this pressure appears to be twofold: power and compatibility. Power in that really quite trivial tasks, such as sorting and printing a list of names, can become major problems in FORTRAN, and compatibility in that even when such problems have been solved the coding must often be changed when transferring the program to another machine. In fact, character handling is probably the major cause of incompatibility between FORTRAN compilers.

The new American National Standard for FORTRAN will resolve many of these difficulties. It is proposed that variables can be declared to be fixed length character strings; the declaration being done by the non-executable statement CHARACTER. Strings may be of any positive non-zero length. Assignment and comparison are performed in the
usual syntax, blank padding or truncation being provided as necessary. A new operator (/) is provided for concatenation, and dissection is effected by a subscript notation. Input or output of strings is still performed by the A-format specification.

2. The need for variable length strings

While the new standard will solve the major problems, there are still some situations which will cause difficulties unless the length of the string may vary.

A variable length string has the great advantage of being more natural and unambiguous to use. For example, in assignment, the programmer need not worry about possible truncation: in passing formal parameters, no care need be taken about matching string lengths; in concatenation, operations to remove unwanted blanks are unnecessary, and so on.

Variable length strings are also useful in that they save storage in the general case. If a list of names is being read into a string array, for example, a fixed length solution requires that all the strings be of a length to accommodate the longest name present, say 'Twistleton-Wickham-Fiennes'. Thus if 'Smith' is stored over 80% of storage is wasted at that location. This high overhead might lead to a serious loss in efficiency due to greater reliance on backing store, or higher cost due to a greater demand for core storage. It also means that the program must be changed if an even longer name should be encountered one day.

The most serious drawback to the use of variable length strings is the inefficiency of addressing them. A fixed length scalar string can be addressed directly, and an element in a fixed length array by a simple mapping function, but variable length strings must involve addressing via a pointer. The necessity for pointers and for other indices such as the text length, typically adds 30% to the storage requirement.

Fixed length strings do not vary in their demand for storage during execution, but when a variable length string is overwritten its storage requirement changes, in the general case. This means that the old text will be abandoned and the new text placed elsewhere in a reserved area of storage. After some manipulation the reserved area may become full whereupon a garbage collection phase is necessary to remove the inactive texts. This operation may be a serious overhead and it is normally impossible to decide in advance upon an optimum size for the reserved area of the storage.

A further disadvantage of variable length strings is that conventional FORTRAN fixed format input/output is not appropriate, as the number of characters to be handled is not normally known before execution. Free format or variable format facilities must be provided, or input/output must be done in an indirect fashion by loading the texts into arrays with blank padding to a fixed length.

Some years ago a data base management problem arose that involved handling lists of equipment to be installed in a hospital. The descriptions of the items could vary quite widely, between say 'Stainless steel sink, 1524 mm x 533 mm, bowl 533 x 381 x 203, left hand drainer, Steristeel AS/SE3' and 'Mop'. It was calculated that the worst case would be about 150 characters and the mean length would be about 40 characters. It also became clear that just the extra effort of moving around non-significant padding characters in fixed length strings would far outweigh the overheads of a variable length solution. Further savings could also be expected from the 70% reduction in core storage and the less frequent disc accesses.

3. Strategy

A set of routines was duly written which used integer variables to index an array dedicated to string storage. This array contained 'string sequences' each of which consisted of the address of the indexing variable, the string length, the text itself and possible padding to the next word boundary; thus each pointer addressed on a word boundary. The padding consisted of zero bits to enable text comparisons to be done word by word.

When a string was redefined, a new sequence was created at the first free location in the dedicated array. The old sequence was then inactive, and this was indicated by setting the pointer address (the first word in the sequence) to zero. When the dedicated array was full, a garbage collection routine was called that took each sequence in turn: all active sequences were moved down the array to overwrite the space released by the inactive sequences. The pointer address was used to change the pointer contents to the new value. The arrangement is shown in Fig. 1.

It was decided to make implementation as machine independent as possible. All but one routine was written in FORTRAN, and input/output was done via integer arrays.

All operations required all the string pointers involved to be passed through a parameter list. Thus assignment, for instance, was done by the statement:

CALL SCOPY (I, J)

The advantage of this scheme is that when a pointer is associated with a new text string its old string can be marked as inactive. If the position of the string being 'overwritten' was not available the only way the garbage collection routine could work would be to have a list of the addresses of all pointers. This implies that pointers would have to be declared before use which would make it more difficult to have pointers local to a subroutine, and would also make garbage collection much less efficient as an additional pass would be necessary to determine which strings were inactive.

It was found in practice, however, that this scheme was rather unnatural and thus error-prone: programmers were continually writing string assignment statements using an equals sign as in:

\[ J = 1 \]

instead of using SCOPY. The effects of this error might well go undetected until the next garbage collection phase, and then only the address of the misassigned variable could be given. It was also found that coding became rather lengthy because only one string operation could be performed per statement.

It was then decided to rewrite the routines as FUNCTION subprograms instead of SUBROUTINES. This implies, as has been said, that the pointer variables must be defined before use. It has, however, the advantage of being more natural to use, because the statement form is now:

\[ \text{〈pointer〉} = \text{〈string expression〉} \]

\[ \text{or} \]

\[ \text{〈pointer〉} = \text{〈pointer〉} \]
and the string expression may have several operations nested within one statement. The drawback is that it becomes more difficult to use variables local to a routine. Also, from an implementation point of view, garbage collection becomes much less efficient due to the necessity for an additional pass and more complicated both because several pointers can index the same string sequence and because there may be sequences that represent intermediate results if the overflow should occur in the middle of a statement.

The new layout of the dedicated array is as shown in Fig. 2: as variables are declared to be string pointers their addresses are placed at the end of the array. The string sequences now consist of the text length, the text itself and possible zero-bit padding to the word boundary. The sequences are added from the start of the array as before.

Again, efforts were made to keep the routines as transportable as possible. Only two short assembler routines were found necessary, the rest were in standard FORTRAN needing at the most the change of a single DATA statement to convert them to a machine of different architecture.

4. Tactics

The two assembly language routines are for an indirect load and an indirect store. That is, given an address they respectively load a value from that address and store a given value at that address. These routines are both very short.

The garbage collection routine is the most complicated, running to about 60 FORTRAN statements. The scheme used is first to sort the list of addresses at the end of the array in the order of the sequences they indirectly address. Shell’s (1959) method is used for the sort.

Each sequence and each indirect address is then taken in turn, unaddressed sequences are inactive, and addressed sequences are moved down the array so as to eliminate inactive sequences. The pointer addresses are used to update the pointers with their new values. The indirect load and indirect storage routines are used in the above processes.

After the pointer addresses are exhausted, any sequences remaining unchecked must be intermediate results. That is, the garbage collection routine has been called in the middle of a FORTRAN statement involving two or more string operations.

In general, there is no way of accessing the pointers to these intermediate results as they will be buried in a stack or held in a register; consequently, the intermediate result sequences cannot be moved within the storage array.

The definition of new sequences thus begins again from the end of the purged sequences, with the intermediate result block being the upper limit. After the intermediate result block there will be a small gap, less than the length of the sequence that caused the garbage collection call, then the list of pointer addresses to the upper limit of the array.

On the available space again being exhausted, a check is first done to see if the intermediate result block may be eliminated. This is done by checking the pointers to see if any of them index a sequence defined since the last garbage collection. If this is so, the intermediate result block is redundant, and sequences may continue to be defined with the start of the pointer addresses again as the upper limit. When this is reached, garbage collection occurs as before.

If the storage space is exhausted, the program terminates with a message.

A routine to move a character string from one location to another is very basic to the package. It is called MBS (‘move byte string’) and is written in FORTRAN, but uses bit manipulation routines to logically AND, OR, complement and shift words. These functions are available on many FORTRAN compilers, or in the system library, although they are by no means universal.

It is very simple to write an MBS routine that deals with one character at a time, but much more efficient, although more complicated, to move a character string a word at a time.

If the bit manipulation routines are not available on the user’s system it would be better to write MBS in assembly language. In many assemblers, such as the IBM 360/370 series, this can be done in essentially a single instruction.

A final low level routine is a function LOC that returns the address of a variable. Again, this function is available on many compilers or system libraries.

It was decided to place the dedicated storage array in named COMMON, together with a variable giving the length of the array. It is the programmer’s responsibility to set up the COMMON statement before string manipulation takes place. Given the above low level routines it then becomes straightforward to construct a character manipulation package.

A list of the routines we chose to implement is given in Table 1, and a brief description of each follows.

1. Subroutine DECSTR
Declares an integer variable to be string pointer. It uses LOC to get the variable’s address and puts it in the address list. A variable giving the length of the list is updated.

2. Function MAKEST
Takes a specified number of characters from an integer array and converts it to a string sequence; it uses MBS. This routine is used for string input and for setting up string constants as in, for example

\[
\text{STAR} = \text{MAKEST}(11^* + 1)
\]

3. Subroutine GETSTR
Used for the output of strings. It places the text of a string into an integer array starting at a specified character position and with a specified field length. The text is truncated, or blank padding is introduced, if the field length does not equal the string length.

4. Function KOMP
Compares two strings and returns a negative, zero, or positive result depending on the relationship. The comparison is done word by word, attention being paid to sign: the zero-bit padding of the string text is to make this possible; using the usual convention that if two strings of unequal length match, the shorter is taken as the lesser. A length comparison after the text comparison ensures correct results even when zero is a valid character, as on the Univac 1100. The internal representation of a character gives the collating sequence and the routine is therefore machine dependent to that extent.

5. Function LNGTH
Returns the length of a string.
6. Function MERGE
Concatenates two strings into one. It uses MBS to create a new sequence. An arbitrary number of strings may be concatenated by nesting a number of calls within a single statement.

7. Function INSET
Extracts a substring given the starting character and the substring length. The routine is simple with the use of MBS, but checks that the starting position or the length are not out of range must be made first.

8. Function INDEX
Finds the position of a substring within a string. It works by extracting each possible substring in turn as a dummy string sequence, by the use of MBS, then doing a word by word comparison. If there is no such substring, zero is returned.

9. Functions LOPHED and LOPEND
Remove, respectively, leading blanks and trailing blanks from a string. Comparisons are done word by word for a nonblank word, then character by character. The routine MBS is used to set up the new sequence.

10. Functions NUMSTR and STRNUM
Convert a real number into a string and vice versa respectively; STRNUM has an error flag as a side effect which is set if the string is not a valid number representation.

   These routines are the longest in the package, apart from the garbage collection routine, running to about 50 statements each. This is largely because it was decided that NUMSTR should be flexible enough to produce a character string of a number in exponential notation if necessary to prevent the string being too long, and in integer form if the number has no fractional part. Consequently STRNUM has to accept number representations in real, integer or exponent notations.

   These routines as written are relatively slow as they work by repeated division and multiplication to handle each decimal digit in turn. However, as this is only done for significant digits (say a maximum of seven, typically) it was not felt to be worthwhile to improve on this algorithm.

11. Function MULTST
Concatenates a string repeatedly with itself. A common use of this routine is to set up long strings of blanks. It operates by the repeated use of MBS.

12. Function IASCII
Given a string and a character position within that string this routine returns the position of the character within the ASCII collating sequence, thus providing a quick and machine independent means of lexical analysis. It operates by using MBS to move the specified character into the low order bits of a zero word, then using the resultant integer to access an array giving the ASCII equivalent for that character.

The result of a string operation is a simple integer that points to the appropriate sequence in the dedicated array and is not cross-referenced elsewhere. This leads to a number of advantages: it is easy for the programmer to define his own string functions; constructs such as

\[ J = I \]

are valid where I and J are declared pointers; a pointer may appear on both sides of an assignment statement as in

\[ I = \text{MERGE} (I, J) \]

and so on. It also means that string pointers are not a special case with regard to local variables losing their value on return from subprograms: the effect is the same as for other variables.

| Table 1 |
|-----------------|---------------------------------|
| Name            | Function                        |
| Visible to the user |                                 |
| DECSR           | Declare an integer variable to be a string pointer. |
| MAKENST         | Convert text in an array into a string. |
| GETSTR          | Get the text of a string and place in an array. |
| KOMP            | Compare two strings.            |
| LENGTH          | Return the length of a string.  |
| MERGE           | Concatenate two strings.        |
| INSET           | Extract a substring.            |
| INDEX           | Find a substring.               |
| LOPHED          | Remove leading blanks.          |
| LOPEND          | Remove trailing blanks.         |
| NUMSTR          | Convert a real number into a string. |
| STRNUM          | Convert a string representation of a number into a real number. |
| MULTST          | Concatenate a string repeatedly with itself. |
| IASCII          | Return the position of a character in the ASCII collating sequence. |

Invisible to the user:

- CGARB: Perform garbage collection.
- INLOAD: Load a value indirectly (assembler).
- INSTOR: Store a value indirectly (assembler).
- MBS: Move a character string.
- AND: Return the logical product of two variables (system).
- OR: Return the logical sum of two variables (system).
- NOT: Return the logical complement of a variable (system).
- SHIFT: Shift a variable arithmetically (system).
- LOC: Return the address of a variable (system).

and the same precautions, if any, against loss of value must be taken, although they always retain their definition as string pointers.

5. Outcome
The routines as described have been found to be easy to use and of great practical use where text has to be handled. A number of programs have been written using the system over a total of three machines and five compilers; which seems to verify its portability.

In general, efficiency has been found to be reasonably good although garbage collection is nine times slower than under the first version implemented. Garbage collection now takes a little under 0.5 seconds on a CDC 6600 to remove 1,000 average-length sequences at a cost of about 20p on the current system used.

On a virtual storage machine, efficiency dropped sharply. The presence of a long array accessed at random along its length was the cause of a great deal of swamping. It is possible that the inefficiency was the fault of the paging algorithm, and that a better algorithm would try to keep the storage array in core for longer periods, but in general this system is unsuited to VS machines.

If overlays are used, pointers lose their value when their overlay is overwritten in core. In principle, by recording overlay numbers and starting and relative addresses, it is possible for the system to run without loss of information. In practice, this is not recommended because it requires accessing the run time overlay table and this not only is relatively inefficient but also greatly hinders the transportability which is one of the principal aims of the package. Instead, the user should keep the storage array and all pointers at the same level in the hierarchy, typically in the root segment.
The routines have been found to be easy and 'natural' to use, although some tailoring was necessary on the input/output routines MAKEST and GETSTR before their present form was decided upon.

It is appropriate at this point to consider the possibility of adding variable length string facilities to the FORTRAN standard. As has been pointed out, this has already been done on a few compilers, and there are clear advantages to the idea. The arguments against such a course are philosophical, but probably valid.

FORTRAN has survived so long, despite the spirited attacks of academicians who would like to see it stamped out, largely because it is not very machine independent and it is still possible to see the machine through the language. This concept of machine visibility allows programs to be tuned to high efficiency and is probably the reason that FORTRAN and COBOL are still so much more widely used than their more powerful modern equivalents such as ALGOL and PL/I. Fixed length strings fit in with this philosophy, but variable length strings with the necessity of indexed addressing, fluctuating use of run time storage and random garbage collection phases do not.

There does not appear to be a clear answer to this question. FORTRAN has existed until now with virtually no string facilities, although admittedly under pressure in recent years; and COBOL users appear to be quite happy with only fixed length strings.

On the other hand, where text lengths vary widely and arbitrarily it becomes difficult, wasteful and inflexible to decide on a fixed upper limit for the text. Probably the best solution is to live with fixed length strings as in the new standard, but to have in the system library such a variable length string package as has been described for use in difficult cases.

References

ACM SIGPLAN Notices, Vol. 11, No. 3, March 1976
ANS X3J3 COMMITTEE (1976). Draft Proposed ANS FORTRAN (X333/76)
GENERAL ELECTRIC COMPANY (1972). Series 6000 FORTRAN Compiler Reference Manual (2200.01)
GENERAL ELECTRIC COMPANY (1973). FORTRAN IV Reference Manual (3102.01)

Book reviews


This considerable tome with attached control evaluation tables bears witness to the significant developments in computer control and audit since the appearance of the first slim volumes in the early sixties followed by Pinkney in this country in 1968. This book (in its second and enlarged edition) is the product of a task force set up by The Institute of Internal Auditors with three partners of Touche Ross & Co (USA) behind whom is an accumulation of many years practical experience.

The book does not set out to be a complete cure to all computer control ills. Neither is it meant to be a reference manual since full comprehension of the later chapters is dependent upon reading certain earlier chapters. It does, however, offer a very practical and sound approach to the analysis and audit of all varieties of controls, both computerised and manual. It also includes a twelve page glossary and a four page list of recommended reading.

The book first covers basic areas including the need for controls to prevent what the authors call 'Exposures and causes'. Various evaluation procedures and charts are introduced. The following sections explain in detail how the approach can be applied to the different elements of a computer system — applications, systems development, and computer operating (processing). There are additional chapters on advanced topics including online systems, minicomputers and data bases. There is also a section on audit management which advises on the problem of recruiting and training computer audit staff. The section on computer abuse is interesting, not the least because it places the Equity Funding fraud computer in a truer perspective than did television and the popular press.

The objective of the IIA task force was to extend an existing limited distribution external audit manual by incorporating the viewpoints of internal auditors, data processing personnel, and top management and, at the same time, to update the book to reflect current technology. In this they have succeeded. Computer Control and Audit is well written and readable. The book is intended equally for all concerned but it is interesting that the authors place systems developers at the head of their list, whilst confirming that the prime responsibility for controls is with the user.

J. A. SHACKLETON (London)

Associations and the closure statement, by M. Rem, 1977; 115 pages. (Mathematical Centre Tracts 76, Delft 14)

Based on an initial concept of Dijkstra, the book expands on two papers on associations written by the author, Feijen and Dijkstra. The author takes as starting point the opportunities for a store with a high level of concurrency that modern techniques, such as associative addressing and large scale integration, have presented to the programming language designer.

Starting from the definition of an assoc as an ordered n-tuple of names' the book introduces the concept and the properties of the associon and in particular the closure statement. After the formal definition of the closure statement, the following chapters deal with such topics as 'the repetitive construct', 'dynamically created names' and 'the cliques of an undirected graph'. The final chapter is interesting, dealing as it does with the various ideas and concepts that have been rejected in the research. The book, another in the Mathematical Centre Tracts series, is well written and particularly well documented, containing a number of examples, an ample bibliography and cross references in the text.

M. P. COATES (Norwich)

Errata

There were three mistakes in the published version of the ALGOL program appearing on p. 184 of the paper by D. J. Evans, 'On the use of fast methods for solving boundary value problems' (The Computer Journal, Vol. 20, No. 2, pp. 181-184). In the second column the first line should be terminated by a semicolon whilst in lines five and six the symbol and should be in bold type.