Efficient String Operations in Xbase++

Efficient String Operations in Xbase++

The idea for writing this article comes from a discussion in the Alaska Software News Groups where the problem of transferring the contents of a text file into an array has been discussed. Several solutions to this problem have been posted in the News Groups.

Although all solutions discussed lead to the same result, the programming task can take many minutes to compute, or it can be achieved within a few seconds. The difference between "many minutes" and "a few seconds" depends solely on how the solution is impemlented.

This article outlines different approaches for solving the same problem and points out strengths and weaknesses of each solution. Each solution discussed in this article leads to the same result. The difference lies in time consumption: a function may take 1, 10 or 100 units of time to finish. Whether a function needs 1 or 100 units of time for completion is highly dependent on how a function is implemented.

The programming task

The task we are going to discuss is pretty simple: read a text file's contents, extract all lines individually and store each line in one element of an array. The array can then be used for further analysis of individual array elements (text lines) or for displaying the file contents in a browser, for example.

The term "text line" is used in this article for a string that ends with Carriage Return+ Line Feed characters (CRLF = Chr(13)+Chr(10)). A text line is not limited to one line of text when printed on paper.

Clipper style solution

Clipper offers Memo functions for processing text, and that's what we are going to start with. MemoRead(), MLCount() and MemoLine() are well known for analyzing a text and extracting individual lines from it. So, the solution is very short and programmed within a few lines of code:

```
01:   cText  := MemoRead( cTextFile )
02:   nCount := MLCount( cText )
03:   aLines := {}
04:
05:   FOR i:=1 TO nCount
06:      AAdd( aLines, MemoLine( cText, ,i ) )
07:   NEXT
```

This code runs perfectly under Clipper and Xbase^{++}. The MemoRead() function is by far the easiest way of loading the contents of an entire text file into memory for further processing. As a side note: in Xbase^{++}, MemoRead() is not only the easiest but also the fastest way.

MLCount() determines the total line count of the text and gives us the loop counter limit in the FOR..NEXT loop. It is line #6 where each line of the text is extracted with MemoLine() and added to an array. And this line is going to produce headaches with Xbase^{++}.

The reason why you are going to stumble over this line is simply the fact, that Xbase^{++} does not limit the length of a string in memory to 64k, as Clipper does. You won't see a problem in Xbase^{++} as long as you process text that would fit into Clipper's memory limits. But if you were using this code to analyze, for example, the contents of your Web server's log-files, you'd be impressed how long your computer can be busy with such a simple task. Depending on file size and hardware, it can take hours until the FOR..NEXT loop is complete.

This runtime behavior comes from the MemoLine() function, which is not only a function for extracting but also for formatting text. A text line can be an entire paragraph consisting of > 1000 characters. Clipper's MemoLine() function was designed for handling this type of text, so that fixed font printer output was made easier. Xbase^{++}'s MemoLine() function works the same way and it does not know if it gets the same text again. It must format the text all over again until the desired text line is identified. Therefore, the time consumption of the FOR..NEXT loop is not linear, but grows exponentially with the length of the text, or file size.

The MemoLine() function is fine and convenient for short text, but you should avoid it when you have to process more than 64k of text. In the following discussion we are referring to the speed of the MemoLine() approach for extracting lines from a 64k text file as 100%. This gives us an idea of what gain in speed can be achieved using different implementations.

Obtaining linear runtime behavior

The first step for optimizing the MemoLine() approach is to achieve linear runtime behavior. This can be done if the text is parsed for CRLF characters and each line is extracted from the beginning of the text string:

```
01:   #define CRLF   Chr(13)+Chr(10)
02:
03:   cText  := MemoRead( cTextFile )
04:   aLines := {}
05:
06:   DO WHILE ( nStart := At( CRLF, cText ) ) > 0
07:      cLine := SubStr( cText, 1, nStart-1 )
08:      cText := SubStr( cText, nStart+2    )
09:      AAdd( aLines, cLine )
10:   ENDDO
```

This solution leads to the fact that the entire text is read only once, which results in a linear time consumption. The DO WHILE loop requires less than 10% of the time consumed by the MemoLine() approach. So, the programming task is completed 10 times faster by choosing a different implementation.

A speed increase by factor 10 is always something worth to think about. However, there is still room for improvement and we're going to see how this code can be optimized further. The "weak" point of this code is line #7 where the extracted text line is "chopped off" the text. To understand why it is a weak point, let us look at a sample text:

```
Step 1:   cText -> "aaaaCRLFbbbbbbCRLFcccccccc"
Step 2:   cLine -> "aaaa"
Step 3:   cText -> "bbbbbbCRLFcccccccc"
```

The variable *cText* holds a string container with 26 characters. When *cLine* is extracted, Xbase^{++} creates a new string container for "aaaa", which ends up in the array *aLines*. In order to extract the next text line, the characters "aaaaCRLF" are removed from the text. This, again, results in a new string container for the remaining characters "bbbbbbCRLFcccccccc" while the original 26 character string is discarded.

Speaking in terms of memory management, this example demonstrates that memory must be allocated within the loop over and over again, that is already allocated. The memory required for the string in Step 3 is already allocated in Step 1. Having this in mind, we can optimize the code by improving the memory management.

Speed it up with the []-operator

In order to improve the memory management for string operations, we have to reduce string operations to the necessary minimum. If it is possible to leave the length of strings unaltered, we should find a way of doing so. Each time the length of a string is changed, new memory is allocated and the original string becomes subject to garbage collection.

Xbase^{++} allows the array operator to be used with strings, i.e. with this operator we can find a single character at a given position in a string. Since we know that a text line ends with a Carriage Return character, it is not necessary to search for the Carriage Return/Line Feed pair, we can program a loop that scans the text for one character only:

```
01:   #define CR  Chr(13)
02:
03:   nStart := 1
04:   nPos   := 0
05:   nEnd   := Len( cText )
06:   aLines := {}
07:
08:   DO WHILE ++ nPos <= nEnd
09:      IF cText[ nPos ] == CR
10:         cLine  := SubStr( cText, nStart, nPos-nStart )
11:         nPos   ++
12:         nStart := nPos + 1
13:         AAdd( aLines, cLine )
14:      ENDIF
15:   ENDDO
```

The logic of this loop identifies the start and end position of a text line and extracts the line without altering the original text. This results in an optimized memory management since we need only one SubStr() call. If we compare the speed of this loop with the previous solution, we find a considerable improvement, i.e. this implementation is about 25% faster than using two SubStr() calls.

Scanning a text string ultra fast

The basic principle for analyzing a text string efficiently is to scan the text string once, to identify start and end position of the substrings of interest, and to leave the original text string unaltered. The array operator is suited to identify single characters in a text string, so that start and end position of a substring can be found. Xbase^{++}, however, offers a better way of searching a substring within a text string: the At() function.

```
01:   #define CR  Chr(13)
02:
03:   aLines := {}
04:   nStart := 1
05:
06:   DO WHILE ( nEnd := At( CRLF, cText, nStart ) ) > 0
07:      AAdd( aLines, SubStr( cText, nStart, nEnd-nStart ) )
08:      nStart := nEnd + 2
09:   ENDDO
10:
11:   IF nStart < Len( cText )
12:      AAdd( aLines, SubStr( cText, nStart ) )
13:   ENDIF
```

The third parameter of the At() function defines the start position for the search. Note that this parameter does not exist in Clipper but is introduced in Xbase^{++} for the very purpose of scanning a text string ultra fast. When a substring is found in a text, the third At() parameter allows for continuing the search after this substring. The offset for the start of the next search is calculated in line #8. This DO WHILE loop runs approximately 4-5 times faster than the []-operator solution.

Squeeze out the last milliseconds

The programming task of transferring a string into an array is now optimized in terms of string operations. The memory requirement for string operations is minimized and the At() function's third parameter is taken advantage of. Compared to the MemoLine() approach with 100% time consumption for a 64k file, we have a solution that needs only 1-2% of the time of the initial implementation. This represents a speed gain by a factor between 50 and 100. A speed gain in this dimension is really impressive and may be satisfying for many developers. But there is still room for improvement!

The optimization that is still possible is marginal when we operate on a file that is only 64k in size. But what if we want to transfer 10 MB, 20 MB or 50 MB into an array? This is possible with Xbase^{++}, it is impossible with Clipper.

A 64k file may contain 1.000 to 2.000 lines of text, i.e. up to now we have dealt with creating an array of up to 2000 elements. A 10 MB file, however, may result in an array of 100.000 or more elements. That means, a really large array is created dynamically, and this is the area for further optimization: the dynamic creation of large arrays.

```
01:   #define CR  Chr(13)
02:
03:   nStart := 1
04:   nCount := 0
05:   nSize  := 100
06:   nStep  := 500
07:   aLines := Array( nSize )
08:
09:   DO WHILE ( nEnd := At( CRLF, cText, nStart ) ) > 0
10:      IF ++ nCount > nSize
11:         ASize( aLines, nSize += nStep )
12:      ENDIF
13:      aLines[ nCount ] := SubStr( cText, nStart, nEnd-nStart )
14:      nStart := nEnd + 2
15:   ENDDO
16:
17:   IF nStart < Len( cText )
18:      IF ++ nCount > nSize
19:         RETURN AAdd( aLines, SubStr( cText, nStart ) )
20:      ENDIF
21:      aLines[ nCount ] := SubStr( cText, nStart )
22:   ENDIF
23:
24:   RETURN ASize( aLines, nCount )
```

This code does not use the AAdd() function for adding each extracted text line to the resulting array, it uses ASize() instead (line #11). The array grows dynamically in chunks of *nStep* elements and this yields a higher efficiency in memory management. When a large array can be expected as result, ASize() is useful to pre-allocate memory for array elements to be filled with strings. When the dynamic array grows to 100.000 elements, for example, it is much more efficient to grow the array 200 times by 500 elements than to call AAdd() 100.000 times.

This final optimization step squeezes out the last millisecond from the programming task of analyzing a text string and transferring each line of the text into array elements. You find the complete source code for the different solutions resolving the task at the end of this document. The test program outputs time consumption of each discussed approach as a percentage. This following output results from transferring a 64k file with 1700 lines of text into an array:

```
1) Clipper MemoLine()    approach        100,00 %
2) Clipper SubStr()      approach          7,87 %
3) Xbase++ []-operator   approach          6,10 %
4) Xbase++ At(,, nStart) approach          1,44 %
5) Xbase++ ASize()       approach          0,80 %
```

Of course, the results of the test program depend on file size, RAM, processor speed and memory used by other applications. The percentage values, however, indicate the magnitude of differences in time consumption (speed) that may result from implementing the same programming task in different ways. The difference is that your program may use 1, 10 or 100 units of time to accomplish the desired task.

Conclusions

The optimization strategies for string operations discussed in this article can make the difference between 1 and 100 units of time a user defined function needs to complete. Except for the At() function's third parameter, all discussed optimizations have to do with memory management, and this is something an Xbase programmer usually does not need to care about.

String operations can put an enormous stress on the garbage collector and can consume a huge amount of memory. For example: it is no problem in Xbase^{++} to load a 10 MB text file into memory with the MemoRead() function. This results in a 10 MB text string that is available in memory. All former "Clipper approaches" to analyse this text string have the potential of performance degradation since Clipper is unable to handle strings of this size.

Let us assume, you have a Clipper function that reads single lines from a text string and this function is rock-stable since many years. This function has never dealt with strings of more than 64k in length! Today you use Xbase^{++}, read a 1 MB text file into memory and pass the resulting text string to your rock-stable function for analysis. To extract the first line of text from a 1 MB text string needs to find the first CR+LF character pair. Your Clipper function uses At() and searches the first occurrance of CR+LF.

The First Occurrance is the limitation of Clipper's At() function. It cannot find the Next Occurrance. In order to find the Next Occurrance, you have to remove the First Occurrance. If the first line contains 50 characters, for example, the first 50+2 characters (text line plus CR+LF) must be removed from the text string. You've done this in Clipper with this function call:

```
cText := SubStr( cText, 50+2 )
```

If *cText* holds a 64k string, this line of code results in 64.000 bytes being discarded to the garbage collector and 63.984 being requested from the memory manager. The parameter passed to SubStr() is discarded and the result of SubStr() is allocated (Clipper is able to handle this situation)

If *cText* holds a 1 MB string, this line of code results in 1.000.000 bytes being discarded to the garbage collector and 999.948 bytes being requested from the memory manager (Clipper is not able to handle this situation)

It is obvious that discarding 64k or 1MB to the garbage collector is a major difference. I.e. rock-stable functions programmed in Clipper can lead to major performance losses in Xbase^{++} when the implicit memory management is ignored and more than 64k memory is required for a single variable or text string.

Although memory management is nothing an Xbase^{++} programmer needs to care about, it is useful to know the implications of string operations on memory management. When the length of a string is changed, the new string requires memory and the original string is discarded. This invokes both, the garbage collector and the memory manager. And both need time to fulfill their job. An optimized programming approach reduces all operations that alter the lenght of a string to the absolute minimum.

Source code for benchmarking

This is the complete source code used to dicuss different optimization strategies for transferring the lines of a text file into an array. You can copy & paste it to your editor, save it as READTXT.PRG and run the benchmark with a file of your choice.

If you own Clipper, you may want to compile this code with Clipper and compare the result with Xbase^{++}. In this case, please note that only two of five implementations are possible with Clipper.

```
* ---------------------- BOF -------------------------
* File: READTXT.PRG

#define CR    Chr(13)
#define LF    Chr(10)

#define CRLF   CR+LF


PROCEDURE Main( cTextFile )
   LOCAL cText, aLines
   LOCAL nTime1, nTime2, nTime3, nTime4, nTime5

   IF Empty( cTextFile ) .OR. ! File( cTextFile )
      ? "USAGE: ReadTxt.exe <textfile>"
      QUIT
   ENDIF

   cText := MemoRead( cTextFile )

   aLines := TxtLine_1( cText, @nTime1 )
   aLines := TxtLine_2( cText, @nTime2 )

 #ifdef __XPP__
   aLines := TxtLine_3( cText, @nTime3 )
   aLines := TxtLine_4( cText, @nTime4 )
   aLines := TxtLine_5( cText, @nTime5 )
 #endif

   ? "1) Clipper MemoLine()    approach", nTime1*100/nTime1, "%"
   ? "2) Clipper SubStr()      approach", nTime2*100/nTime1, "%"
```

```
 #ifdef __XPP__
   ? "3) Xbase++ []-operator   approach", nTime3*100/nTime1, "%"
   ? "4) Xbase++ At(,, nStart) approach", nTime4*100/nTime1, "%"
   ? "5) Xbase++ ASize()       approach", nTime5*100/nTime1, "%"
 #endif
RETURN


/*
 * Standard Clipper approach using MemoLine()
 */
FUNCTION TxtLine_1( cText, nElapsed )
   LOCAL nTime := Seconds()
   LOCAL i, nCount, aLines

   aLines := {}
   nCount := MlCount( cText, 254 )
   FOR i:=1 TO nCount
      AAdd( aLines, Memoline( cText, 254, i ) )
   NEXT

   nElapsed := Seconds() - nTime
RETURN aLines


/*
 * Clipper approach replacing MemoLine() with SubStr()
 */
FUNCTION TxtLine_2( cText, nElapsed )
   LOCAL nTime := Seconds()
   LOCAL aLines, cLine, nStart

   aLines := {}

   DO WHILE ( nStart := At( CRLF, cText ) ) > 0
      cLine := SubStr( cText, 1, nStart-1 )
      cText := SubStr( cText, nStart+2    )
      AAdd( aLines, cLine )
   ENDDO

   nElapsed := Seconds() - nTime
RETURN aLines
```

```
#ifdef __XPP__

/*
 * Xbase++ approach using the []-operator with strings
 */
FUNCTION TxtLine_3( cText, nElapsed )
   LOCAL nTime := Seconds()
   LOCAL aLines, cLine
   LOCAL nStart, nPos, nEnd

   nStart := 1
   nPos   := 0
   nEnd   := Len( cText )
   aLines := {}

   DO WHILE ++ nPos <= nEnd
      IF cText[ nPos ] == CR
         cLine  := SubStr( cText, nStart, nPos-nStart )
         nPos   ++
         nStart := nPos + 1
         AAdd( aLines, cLine )
      ENDIF
   ENDDO

   IF nStart < Len( cText )
      AAdd( aLines, SubStr( cText, nStart ) )
   ENDIF

   nElapsed := Seconds() - nTime
RETURN aLines

/*
 * Xbase++ At() function with third parameter
 */
FUNCTION TxtLine_4( cText, nElapsed )
   LOCAL nTime := Seconds()
   LOCAL aLines, cLine, nStart, nEnd

   aLines := {}
   nStart := 1

   DO WHILE ( nEnd := At( CRLF, cText, nStart ) ) > 0
      AAdd( aLines, SubStr( cText, nStart, nEnd-nStart ) )
      nStart := nEnd + 2
   ENDDO
```

```
   IF nStart < Len( cText )
      AAdd( aLines, SubStr( cText, nStart ) )
   ENDIF

   nElapsed := Seconds() - nTime
RETURN aLines


/*
 * Xbase++ memory optimization with ASize()
 */
FUNCTION TxtLine_5( cText, nElapsed )
   LOCAL nTime := Seconds()
   LOCAL aLines, cLine, nStart, nEnd
   LOCAL nCount, nSize, nStep

   nStart := 1
   nCount := 0
   nSize  := 100
   nStep  := 500
   aLines := Array( nSize )

   DO WHILE ( nEnd := At( CRLF, cText, nStart ) ) > 0
      IF ++ nCount > nSize
         ASize( aLines, nSize += nStep )
      ENDIF
      aLines[ nCount ] := SubStr( cText, nStart, nEnd-nStart )
      nStart := nEnd + 2
   ENDDO

   IF nStart < Len( cText )
      IF ++ nCount > nSize
         RETURN AAdd( aLines, SubStr( cText, nStart ) )
      ENDIF
      aLines[ nCount ] := SubStr( cText, nStart )
   ENDIF

   nElapsed := Seconds() - nTime
RETURN ASize( aLines, nCount )

#endif
* ---------------------- EOF -------------------------
```