Clarion ties a Memo Array by size to a specific output screen and/or report. This becomes a problem when editing in a different-sized window or when one needs a special report. This article details two aspects of generic text storage and how to implement them using Clarion's Memo Arrays.

Clarion, by default, assumes that a person will use a Memo Array with a specific size output that is the same size as the STRING and DIM of the Memo Array declaration. For the most part, this is a valid assumption and works well with Designer-generated code. It does not, however, allow any flexibility and specifically it does not allow one to use a Memo Array as a generic storage bin for character-based free form text.

In this article I will deal with two aspects of generic text storage and how to implement them using Clarion's Memo Arrays. The first is how to edit a Memo Array from any size screen, and the second is how to output a Report that is of a different String length than the width of the Memo Array.

Editing a Memo Array

To fully understand what the problem is, we must first examine how Clarion stores Memo Arrays. When working with Memo Arrays in a Screen one usually uses the TEXT() and USE() statements in the screen to declare the Memo Array editing field. When used, one of the shortcomings of this statement combination quickly becomes apparent. As mentioned on pages 8-18 of the [2,1] Clarion Professional Developer Language Reference Manual:

"Because of word-wrapping, USE variables of TEXT fields become 'width sensitive'. If a width sensitive variable is used for a TEXT field with a different width, unnecessary spaces will appear between words, and words will span rows."

What this means to you as the user is that if you define a TEXT() field in a screen that is of a different size than the TEXT() field under which the MEMO was originally entered (with word wrap set on), the MEMO will not appear as one would expect because Clarion added spaces to the original MEMO to pad out each line on the screen to effect its word-wrapping. If the MEMO was originally input into a TEXT() field without word wrap and is then brought into a TEXT() field with word wrap set on, then
Clarion will display the MEMO by word wrapping each line and introducing spaces at the end of each line. Once the spaces are added, they become part of the MEMO.

No problem so far. After viewing and making any changes, if the screen is ACCEPTed with [Ctrl-Enter], Clarion saves the TEXT() field back to the MEMO literally, including the blanks it added at the end of each row (again, assuming word wrap is set on). If you only use the MEMO by viewing it in one screen or from screens of the same size, you may say "Who Cares?", but as soon as you try to use it in a screen of another size or try to make a Report with lines of a different length, you will say "I care!". Each of the added blanks now fall not at the end of the line but somewhere in the middle of the line with less than aesthetically pleasing results. These added blanks are the root of the problem in trying to use a Memo Array in any generic way. Fortunately, it is easy to cure using my blank-stripping procedure.

When we examine the map of the example program, named StrpExmpl.CLA, we find four procedures. You will recognize G_OPENFILES as the standard file opener generated by Designer. MAIN is a simple ACCEPT loop which allows us to edit one of the fifty-two memo fields provided by means of a call to Edit_Text. (The number fifty-two is chosen only to match the number of entries I have in MASTER1 for this example program.) Edit_Text is a simple Clarion PROCEDURE with a few modifications such as the PLZ_WAIT screen. We must pause here to examine the MEMO and TEXT() fields. Notice that in the definition of the data file MASTER1 we have:

```
Mas::Text           MEMO(840)
```

and the normal array to hold this memo defined as:

```
MasMemoRow          STRING(40), DIM(21)
```

over it.

But in the Edit_Text Procedure, located in the SCREEN code, I have deliberately set

```
TEXT(14,60), USE(Mas::Text)
```

statement to be a different row/column size. So, in the Edit_Text procedure we see that it displays the MEMO field in the fourteen-by-sixty screen and allows simple editing. When the ACCEPT_KEY is pressed, ?Mas::Text field is completed and processing falls through to ?LAST_FIELD where we open the PLZ_WAIT screen and call Blnk Strip(). The call to Blnk Strip() with its INTs and SIZE and MAXIMUMs may look like a lot of extra stuff but it provides for a generic call that is independent of how you size your arrays. I have put in a commented equivalent call using the actual numbers to help you decode what is happening. (This decoding is left as an exercise for the student. See Chapter 13 in the Clarion Professional Developer Language Reference Manual.)

Blnk Strip() is where the real work gets done. The passed parameters are explained in the comment header of the Procedure. They are in general the input/output array names, string length, and dimension size. I have also defined a string variable, Holder, at the maximum string length. This does limit the maximum Array String Length allowed in either the Input or Output arrays, but it is unlikely that one would ever have a screen of that size anyway. The first loop back-steps by rows in the Input_Ary[] to find the first row containing text, or the last row containing text depending on whether you view a glass of water as half empty or as half full. We come out of this loop with MemoInSize set to that row. We now initialize all the pointers.

The next loop takes us through the Input Array rows until all rows that have text are processed or until the Output Array is full. Nested inside this loop is the loop that examines each string for excessive blanks. InStrPtr, the INSTRING() pointer, is loaded by the INSTRING() function with the location of the first occurrence of a double blank. Notice that we have a StartNo variable that says where to start the INSTRING() search. On the first pass, it is set to one so it will locate the first double blank in the string. If InStrPtr is loaded with

---

**Publisher's Note** – The source code for this article is quite lengthy. Please reference either the source files on your CTJ code disk or download the code from PCIG Information Exchange. By reducing the excessive code listings (> 6-8 pages per article) we have additional space to publish subjects that are more diverse and cover a broader range of Clarion-related topics, product reviews, etc.
zero by the INSTRING() function then no double blanks were found and we may add the string to Holder.

Examine the nest of IF statements starting with the statement "IF NOT InStrPtr". The outside IF branches depending on the returned value of InStrPtr. If InStrPtr is FALSE it enters the first branch. The next IF branch depends upon the value of NeedBIFlag, the Need-A-Blank Flag. This part gets ugly. Consider that the program must deal with the following cases:

1. The string starts with double blanks (DBs).
   Then they must ALL be stripped out.
2. The string starts with only one blank. Then it must be left in.
3. The string starts with a character. Obviously, no blank needed here.
4. The string has DBs somewhere in the middle. Only one blank must be left.
5. The string ends with DBs. Here one blank must be left in. Notice that this decision is dependent upon our choice made in 1) where we strip ALL the blanks at the beginning of the string. Therefore, if we have one string ending with DBs and the next also starting with DBs, we leave in a blank at the end of the first string. This is an arbitrary decision and could just as well have been the other way around as long as we remain consistent.
6. The string ends with one blank; leave it in.
7. The string ends with a character. Again, no blanks needed.

Whew! Did I miss anything? I hope not! Looking again at the NeedBIFlag-IF nest, we know that we are not dealing with DBs anywhere since we are in the NOT InStrPtr branch. Therefore, we know that we are clear to the end of the string and there are only two choices. The first choice is that we have previously put something in Holder, the temporary string holder, and we need a blank to space out the next string to be added. The line "Holder = SUB(....)" does this. Notice that we CLIP() Holder since it is actually 225 characters long and Clarion will add the new text on the end unless we CLIP off the trailing Blanks. (Ah-Ha! So THAT'S why he has that NeedBIFlag!) In the NeedBIFlag = TRUE branch, where we are, we now add in a blank and then the new text. The SUB() function extracts the text from the Input_Ary[] beginning at StartNo position and ending with In.Len. (Note that we can use In.Len as the terminator for the SUB() function since we are in the NOT InStrPtr branch.)

We need to make one more check for the case where string is DB free (a DB-Free Zone?), BUT it ends with a single blank. In this case we need to set the NeedBIFlag to TRUE so that a blank will be added in when we start the next Input_Ary[] row. The insidemost IF does this check, and sets NeedBIFlag to TRUE or FALSE depending upon the last character in the string. The second choice, remember we had two choices, is that no blank is needed. The code starting with the ELSE statement deals with this. It is a duplication of the code we just went through except that we do not add a blank after we CLIP() Holder. The last line in this NOT InStrPtr branch is to set the StartNo to the maximum plus one. This will flag the reading in of a new Input Array row further along in the code.

The next branch starting with ELSE is the case in which InStrPtr has a value other than zero, indicating that some DBs have been found. The value in InStrPtr indicates the starting position in the Input Array string of the DBs. We have two more IF chains here. The first chain checks two conditions. The first condition is for DBs found at the beginning of the string. In this case, InStrPtr will be the same as StartNo and we will do nothing since there are no characters before the DBs to put into Holder. The ELSIF and ELSE branches you will recognize as similar to the ones in the NOT InStrPtr branch. Notice, however, that here the terminator for the SUB() command is now InStrPtr. If we wanted to be precise, it would be "InStrPtr - 1" to drop the blank, but since we do a CLIP() of Holder before adding more text it isn't needed. The next IF chain sets NeedBIFlag. Normally it will be set to TRUE since we have just found a string of blanks that will need to be removed. There is one case in which StartNo equals InStrPtr AND StartNo = 1. This corresponds to our case 1), in which there are DBs at the beginning of the string and all need to be removed. In this case NeedBIFlag = FALSE.

The next process is to step forward from the found DBs and find the next non-blank character.
To do this we set StartNo to the next character, which is InStrPtr + 1. The LOOP then extracts each subsequent character into Temp and examines it for a non-blank condition. If it is blank, we step StartNo and check again. When a non-blank is found OR we run out of characters in Input_Ary[], we break out of the LOOP. At this point, StartNo is pointing to the next non-blank character or is out of bounds. If out of bounds we step to the next Input Array row and reset InStrPtr to one. We then loop back and do the inner LOOP again. This process continues until we have enough in Holder to make up an Output row. In that case, we BREAK out of the inside LOOP and transfer a row to the Output Array. We then rearrange Holder with the SUB() function so that the remaining characters are all rippled down to the beginning of the string. Last of all we bump the Output Array pointer OutArrPtr by one and go do it all again.

Eventually, we will run out of Input Array rows or Output Array room and fall out of the outside LOOP. There are still two cases to check for, both involving text still remaining in Holder after the last Output Array row has been handled. If there is still room in the Output Array, we just transfer the contents of Holder to the output array. We could CLIP() this last row, but Clarion does this for us when we PUT the Memo field—so it is not necessary. In the case where we have exceeded the Output Array, that is a problem of not having made the array STRING/DIM sizing large enough and in the words of whoever it was, “That’s not my problem!” Just resize the Output Array properly and you will be OK.

When we return from BInk_Strip to the calling Procedure, Edit_Text, we must transfer the Output Array back into the original array. I do this with a LOOP and a row-at-a-time transfer since the two arrays are the same size. In most cases the Input Array and the Output Array will be the same size, but I have made the BInk_Strip Procedure as generic as possible because there are cases when the arrays may need to be different. For that case the easiest way to transfer the stripped array back is to have made Out_Ary[] OVER some other MEMO() field similar to the way we set up MasMemo_Row. This then allows you to just equate the two MEMO labels and the text will be transferred. This may be faster for MEMOs of very large size.

The observant reader will notice that this Procedure is not only good for stripping blanks, but is handy for moving information between different sized arrays in general. There is only one problem I can find with this entire process. That is, you may want some of the blank areas inside the MEMO lines and only want the DBs at the end removed. The only way I know to do this is to back-step in each string removing blanks from the end only. I haven’t done this because for most cases I need to have all the doubles removed. You may not know if you have extra blanks in your memo fields. To determine this you can look with Scanner using the Control-T hot key to pop up the Memo Text. In general, you can assume that you do have blanks since this is the Clarion default unless you start right out using this blank stripping procedure. It is probably safest to just include the call to BInk_Strip() in most of your programs that deal with memes. All you lose is a little processing time. So the next time you look with Scanner and see strange accumulations of blanks in your Memo fields, use this procedure to tidy up those fields and keep them clean.

**Memo Field Output**

For our example of Memo Field output I will use a Report case from our office. When it is necessary, as in our case, to keep on hand hardcopy of large data sets, it becomes desirable to produce such reports in compressed-print format. This usually is the eight- and-one-half point, sixteen-pitch Line Printer font since it is a built-in font in most Laserjet printers, but it could be any cartridge or downloadable font. When a report includes information contained in a Clarion Memo field, it can become complicated to reformat the information to fit the report line. Particular attention must be paid to word wrapping to avoid large gaps or partial words at the end of a report line.

What I use is a procedure Get_Memo_Ary(), that takes in a memo array and returns another array of the proper size for the desired report output. In the example MemoArray program used here, the main data file is Master1. Notice that it has the usual structure for a Memo field
with Mas:Text being the MEMO(840) and MasMemoRow with STRING(60), DIM(14) being the GROUP over Mas:Text. (This is the same data file we used in our blank stripping example.) The CODE section makes a call to Designer's standard G_Open Files and then calls our example report PROCEDURE, REx ample. There are three REPORT statements — two of which deal with the changes in pitch and point size. Each of the CONTROL USE statements sets variables which contain HP Laserjet II escape sequences. These command strings control the printer font and line spacing settings. I prefer to make these separate REPORTs rather than DETAILS in one Report so that the line count will not be effect ed. (This is also a useful technique for printing multiple headers on the first page but requires that you keep track of your own line count for page overflow.)

Editor's Note: I added two more REPORT statements to take the place of the Laserjet escape sequences. These two additions perform the same function, but are for Epson compatible dot-matrix printers. The appropriate PRINT statements (2) for each are grouped together at the beginning and end of the procedure. Depending on what kind of printer you have, you comment out the other statement.

In the third REPORT there are two DETAILS of interest, one that is a publication number and the other a related Memo of size STRING(100). The task is to convert the original MasMemoRow array of length 40 to an output string of size 100. To do this we define a holder array Out_Ary STRING(100),DIM(9) which is large enough to contain all of the MasMemoRow array. Once the information is in this array, it can be read a line at a time directly from the Out_Ary to Detail2 and printed.

The CODE section does some flag and pointer housekeeping and opens a simple "Please Wait" screen. The first print statement, PRINT(F16: Rpt_Head), sets the printer to Line Printer sixteen-pitch font and eight lines per inch using the Set variables Set_16LPtr and Set_8LP. (The detailed explanation of these HP LaserJet codes is beyond the scope of this article but they can be decoded using the Printer Command Tables located in the appendix of the HP LaserJet Users Manual. Similar escape sequences can be created to control dot matrix printers. After a simple SET() command using the publication-number KEY, we enter the main LOOP I am using a flag called DoneFlag here to allow inclusion of the Next_Record ROUTINE.

Strictly speaking, the Next_Record ROUTINE is not needed here. The code is so simple that it could be inline, but I include it for the sake of consistent form. DoneFlag will be set to TRUE by the Next_Record ROUTINE when Master1 is at End-Of-File.

The call to Check_Page is necessary because I am handling line count and page overflow. Clarion is getting better at this but I find that it is still better to do it myself for non-standard reports. Check_Page looks to see if Mem:Line, the built-in Clarion {Editor's Note: built into Clarion Designer} line-counter Memory variable, is greater than 75. This is based on eight lines per inch on an eleven inch page which is 88 lines total. Of course, there is some top and bottom margin and I am also making sure that I do not run out of page if I get a Memo field that is six or seven lines long starting on line 74. After a check for End-Of-Page the first DETAIL is printed.

I need to comment on this first DETAIL, the two Rpt_Head DETAILS, and the Form_Feed DETAIL, as they are slightly non-standard. Notice that they DO NOT contain a command CTL(@LF) such as you see in the MemoLine DETAIL. When you create a Report with Reporter, it automatically puts in this command at the end of each DETAIL which generates a Line Feed to the printer. We do not really need a Line Feed with the Rpt_Head details and we explicitly do NOT want one in Detail1 and Form_Feed. Detail1 prints the Publication number from column eleven to column fifteen and puts a dot in column 16. I then want one blank and the start of the Memo without any Line Feed. Detail2 is designed to start in column eighteen. This then gives an output with the Memo indented and the Publication Number as a Hanging indent leader.

The next step is to call the Bnk_Strip() Procedure which we previously discussed. This is
necessary here for the same reason it was necessary with the Screen example, since the report is of a different size than the original Memorow declarations. If we rearranged the output row length without calling it we would end up with extraneous blanks in the middle of our output lines, corresponding to the previous End-Of-Lines in Memorow. On return from Blnk_Strip, I clear out the original array and transfer the now-stripped array back into the original array. This is not necessary, but it is easier to follow the program logic if we don’t change the names of the players in mid-game.

Now the call to Get_Memorow is made and we can examine where the real work is done, in the Get_Memorow PROCEDURE. Notice the passed parameters. These are explained in the procedure heading. The first parameter is the passed array. In our example this is Memorow[] from Master1. The next two parameters are the string length and dimension of the input array. The fourth, fifth, and sixth parameters are similar but are for the output array. You will notice that I have maintained a similar structure and naming as I used in Blnk_Strip Procedure. The local versions of the arrays are declared as EXTERNAL, DIM(1) since they are already declared in the calling program. Holder is the string that will be used for holding each input array line during transfer.

After cleaning out the output array, the first LOOP back-steps to find the first non-blank line in the input array. We come out of this loop with MemoinSize set to the last row of the input array that contains data. Next we initialize all the pointers and enter the LOOP nest. The outer loop makes sure that we go through all the input array lines. It also checks to make sure we have not exceeded the size of the output array. This should never happen if we have sized our arrays correctly when declaring them. The next LOOP loads the Holder string until it has enough characters to fill an output array line. We must also check here that we don’t exceed the input array size and that the input array pointer, InArrPtr, gets bumped as each line is read. HolderPtr is incremented each time to point to the last character in Holder. As Holder get jammed full of input lines, it will eventually exceed the output line length, Out_Len, and fall through the first inner LOOP.

The next two LOOPS find the word boundary at the end of the line. The first LOOP back-steps to find the first blank character, decrementing BlankPos1 as it goes. The SUB() function is used to examine characters one at a time. The second LOOP keeps back-pedaling until it finds the next non-blank character. This leaves us with BlankPos2 set to the last letter of the last full word in Holder that will fit into the output array string. This SUB-string of 1-to-Blank_Pos2 length is then transferred into Out_Ary[]. The remaining characters then ripple down to the beginning of the Holder string and HolderPtr is reset by subtracting the same number of characters as are thrown away in the ripple down process. Notice that we throw away the extra blank that would be between the words with the “BlankPos2 + 1” statement since we do not want it at the start of the next line. (Holder could also be used as a revolving buffer if desired, but the ripple-down method is a bit cleaner logically for first-time users.) Last of all we bump the output array pointer (OutArrPtr), and do it all again until all input characters are transferred.

Back in the calling procedure, RExample, we use a LOOP to back-scan the returned Out_Ary[] again to find the last row that has anything in it. MemoRows points to that row when we fall out of the LOOP. Now in the last LOOP we can simply unpack the output array line at a line into the report Rpt:MemoLine of Detail2 and PRINT() that DETAIL. I put in a Line Feed between detail groups and a Form Feed and Font reset at the end of the report. This is the basic form. There are undoubtedly other improvements that can be made to speed processing and/or improve clarity but this should help you well along your way to custom-made Reports of any size using any Clarion Memo Array as a source.

**About The Author**

Alan A. Dunwell is the Senior Software Specialist for JLLA, a research laboratory, and acts as co-host of the Boulder area Clarion Users Group. He has been programming in Clarion since version 2006 and does extensive Clarion hand coding but is slowly being swayed by the Lobby person for the “Designer Special Interest Group.”