Increasing Script Performance, Part II

Why Some Things Are Faster Than Others, Part II - by Wil Dijkstra

Suppose you have a list of numbers, for example a list that is read from a datafile. Common ways to separate the different numbers are CRs, commas or spaces. To get a particular number, in case of a comma-delimited list, you may use something like:

script A:

```
get item 900 of myList
```

In case of a space-delimited list, you may use:

script B:

```
get word 900 of myList
```

Which script is faster? The answer is that script A may be 50 to 100% faster than script B (depending on the size, that is, the number of characters of the numbers in your list). The same logic as discussed in part 1 of "How to speed up MC" applies. To get a particular item from an item-delimited list, the computer has to count commas. That means that for each character in the list, the computer has to decide whether it is a comma or not, until (in script A) 899 commas are counted. In script B the computer has to decide whether a character is a space, a tab or a CR: all three characters are word delimiters. If your list is space-delimited, change script B to (of course your list should not contain spaces):

script C:

```
set itemDelimiter to space
get item 900 of myList
```

and it will be (nearly) as fast as script A. The statement "set itemDelimiter to space" hardly takes time (and of course you will put it outside any repeat loop).

It is easy to understand now that getting word 100 from `myList` will take much less time then getting word 900. Getting the last word takes the most amount of time. If you know your list has 1000 numbers, it may be good to know that:

```
get word 1000 of myList
```

will be faster than

```
get last word of myList
```

The reason is that the computer first has to figure out how many words `myList` contains. Hence,

```
get last word of myList
```

is equivalent to something like:

```
put the number of words of myList into n
get word n of myList
```

Now suppose you have some text, simply consisting of characters, let's say 5000 characters. Which script will be faster?

script D:

```
get character 5000 of myText
```

script E:

```
get last character of myText
```

Contrary to what you may expect now, script E is as fast, or even faster than script D. A variable like myList is stored somewhere in memory. The computer (or MC) has to know (a) the position of the start of the variable in memory and (b) the length (the number of characters) of that variable to prevent that other variables are written over existing ones. To get a particular character, say character 345, the computer just has to add 345 to the starting position of the variable, to know exactly where character 345 resides. Consequently, unlike items, words or lines, all characters in a text are accessed equally fast, and, even more important, very fast.

How can we use this property to speed up our scripts? Let's go back to the example of getting the last word, item or line of a list. Suppose our list is CR delimited. To get the last line, we could use:

script F:

```
get last line of myList
```

However, script G will usually be much, much faster!

script G:

```
put the number of chars of myList into nChars
repeat with i = nChars down to 1
  if char i of myList = cr then exit repeat
```

```
  end repeat
  get char i + 1 to nChars of myList
```

Similarly, to get the one but last line, we can extend our script a bit, as a fast alternative to `get line -2 of myList`. We can use this principle to write a function handler that is similar to the lineOffset function, but searches backwards, to find the last instance of number x in myList. The simple, but slow function, would look something like this:

script H:

```
function scanBackSlow aLine, aList
  put the number of lines of aList into nLines
  repeat with i = nLines down to 1
    if line i of aList = aLine then return i
  end repeat
  return 0
end scanBackSlow
```

Script I is much faster however:

script I:

```
function scanBackFast aLine, aList
  put the number of chars of aList into nChars
  put 0 into count
  put nChars into k
  repeat with i = nChars down to 1
    if char i of aList = cr then
      add 1 to count
      if char i + 1 to k of aList = aLine then
        return the number of lines of aList + 1 - count
      end if
      put i - 1 into k
    end if
  end repeat
  return 0
end scanBackFast
```

Generally you write function handlers that are repeatedly used in your scripts. Unfortunately, calling a handler, passing parameters and returning the result, takes time. Consider script J:

script J:

```
function myFunction n
  -- do something with n
  return n
end myFunction
```

In this example, the handler performs some transformation on n and returns the result. To call the handler from another script, you may write: `put myFunction(m) into m`. In executing this handler, the variable m is copied into variable n, some transformation is performed on n, and the result is sent back and copied into m. Alternatively, you can write a message handler like:

script K:

```
on myHandler @n
  -- do something with n
end myHandler
```

(Note: This is also called "pass by reference"; for more info, see tip scrp003 - The Beauty of "Pass By Reference".)

To call the handler from another script, just write: `myHandler m` to obtain exactly the same result. The difference is that no copying takes place. The transformation takes place on the variable that resides on the place of variable m in memory: no new variable n is created. And hence the handler will execute faster (about 50% if "do something with n" wouldn't take time, but the relative gain depends of course on the time the transformation itself takes). The variable n in `myHandler n` should not be something like `myHandler item 3 of myItems` because `item 3 of myItems` is not a variable (but part of a variable): MC has no idea where `item 3 of myItems` starts, or what its length is. If you don't want to put the result of the transformation of m into the same variable m, there is nothing wrong with script L:

script L:

```
on myHandler @n, @r
  -- do something with n and put the result into r
end myHandler
```

and you can call the handler with:

```
myHandler m, p
```

The result of the transformation of m will be put into p, which will be faster then

```
put myFunction (m) into p
```

In the same way, you can make scripts like script I above, a bit faster if you just write:

```
function scanBackFast aLine, @aList
```

You save the time it takes copying myList into aList.

Happy programming,

Wil Dijkstra

Posted on 4/7/03 by Will Dijkstra to the MetaCard list (See the complete post/thread)

Print this tip
