Converting Between PostScript Strings, Integers, Arrays, and Dictionaries

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552
copyright c2003 as GuruGram #30
http://www.tinaja.com
don@tinaja.com
(928) 428-4073

The general purpose **PostScript** computer language is both richly and strictly **typed**. What may work out well for one data type may be difficult or inconvenient in another. Conversion between types can end up anywhere from being **redbook** obvious to ending up extremely subtle and obscure.

What I thought I'd do here is update and expand our older **STRCONV.HTML** type conversion tutorial and add a few sneaky new routines...

Extracting a Substring

This is easily done by using **(mybigstring) 3 2 getinterval**. Which, in this example gets you a **two** character substring that starts on the **fourth** (counting 0, 1,2,3) character from the beginning. You can also do a **search** that returns **post match pre true** if found or **original_string false** if not. Note that the **left** (or the **early**) portion of the string ends up on the **top** of the stack.

Also note that use of **getinterval** returns a **substring**, while, as we'll shortly see, **get** returns an **integer** equal to the selected **ASCII** character value.

Merging two Strings

PostScript strings are always of a fixed and predefined length. Short of writing over nulls, I know of no immediate or direct process for adding characters to the end of an existing string. Instead, merging two strings takes a custom routine found in my **Gonzo Utilities**...

```
/mergestr {2 copy length exch length add string dup dup
4 3 roll 4 index length exch putinterval 3 1 roll exch
0 exch putinterval} def

  (This is a) ( merged string.) mergestr --->
  (This is a merged string.)
```

Merging is also useful for formatting or to add carriage returns or punctuation. A **(yourstring) (.\n\n) mergestr** adds a period and double newlines should you later decide to do a **print**. Be careful when merging strings to make sure there are just enough spaces between your merged words!

Inserting a Substring

This is easily done by using **(mybigstring) 2 putinterval**. Which, in this example inserts a substring of any length that will start on your **third** (counting 0,1,2,) character from the beginning.

The original string **must** be long enough to hold the inserted one.

Note that use of **putinterval** inserts a **substring**, while, as we will shortly see, **put** inserts and converts an **integer** equal to the **ASCII** character value.

String to Integers

A **mystring 6 get** will extract the **seventh** character in a string, convert it to an 0-255 integer, and place it on the stack. The integer value will equals the ASCII character value.

To convert all of the string into integers on the stack, simply use a **mystring {} forall**. Note that our "empty" **forall** proc will place the integers on the string in correct order.

Note that the integers need **not** be a rational ASCII message. Strings can instead be used to efficiently store **any** data values within an 0-255 integer range.

String to Array

To convert a string into a stack top array of 0-255 ASCII equivalent integers,

```
/mynewarray mark (yourstring){}forall ] store
```

Once again, the "empty" **forall** proc piles up the integers onto the stack for you.

Inserting Numbers Into a String

Here is how you put a **PostScript** integer or real number into a string...

```
/secretnum 42 store

(The meaning of life is )
secretnum 10 string cvs mergestr (.\n) mergestr
```

Strings can be written to a disk using **yourfile (yourstring) writestring**.

Converting Between ASCII and BCD Numbers.

The ASCII code for the numeral "0" is 48. Numeral "1" is 49 and so on...

To convert FROM a 0-9 numeric to an ASCII integer,
ADD decimal 48. (or hex $30)

To convert FROM an ASCII integer a 0-9 numeric to ASCII,
SUBTRACT decimal 48. (or hex $30)

Any longer groups of numbers can be extracted from a string by use of **(1.2345) cvr**. Or else by creative use of PostScript's obscure **token** command.

Converting Between Strings and Names.

Just use the **redbook** solutions of **(string) cvn** or **/Name 10 string cvs**. Note that the forward slash gets added or removed automagically.

String Dereferencing

One of the most maddingly infuriating happenings in all of **PostScript** occurs when the contents of old strings mysteriously change later on in unexpected ways. The usual problem is failing to understand that...

PostScipt does NOT place strings into arrays
or other data structures!

All it places are POINTERS to those strings!

Thus, if you reuse a string (such as a disk reading **workstring**), much if not all of your previous string storage can get corrupted. Note further that...

PostScript string contents are NOT preserved
or protected by saves and restores!

The subtle but simple cure for these hassles is called **string dereferencing**. Any time you are about to use a string that may change later in your program, you take a "snapshot" that creates a **new** and unique string that will not change...

(reusable_string) dup length string cvs ----> (safe_string)

Pointers to your dereferenced "safe string" can now be placed in an array or whatever. **(safe_string)** will stay exactly the way you created it. And your new **(reusable_string)** is free to go on to other tasks.

String to Executable Code

This one is really easy...

```
(yourstring) cvx ---> makes string executable

(yourstring) cvx exec ---> actually executes proc
```

String to Windows Filename

The "\" reverse slash is reserved in **PostScript** strings as an "escape" character. For instance, a **\n** forces a newline, while **\(** or **\)** can be used to deal with unmatched parentheses **within** a string.

Should you really want a \ in any PostScript string, you'll have to insert a \\ . One important place this shows up is in Windows Filenames. Per this rule...

ALWAYS use \\ anytime you want a \ in a PostScript string!

A typical Windows filename string might initially get entered into a PostScript string as **(C:\\windows\\desktop\\gonzo\\gonzo.ps)**.

Gonzo Fourslashing

My **Gonzo Utilities** will "double up" on the reverse slash hassle...

ALWAYS use \\\\ inside a Gonzo text justification any time you want a \ to end up in any PostScript string!

A typical Windows filename string might initially get entered into **Gonzo** text to be justified as **C:\\\\windows\\\\desktop\\\\gonzo\\\\gonzo.ps**

Adding Elements to an Array

PostScript arrays are normally of a **fixed** size. Except for overwriting nulls, I know of no way to directly make an array larger. Instead, use this dynamic scheme...

```
/myarray mark myarray aload pop new_element ] store
```

Since freely mixed types are permitted in PostScript arrays, the **new_element** can generally be most any string or numeric or proc or whatever of your choosing. But if the array is later to become a string, then **new_element** must be an appropriate integer in the 0-255 range.

Convert an Array into Integers

A simple **aload pop** fills the stack with an array's contents. If the array consists of all integers, then the stack fills with the same. That **pop** gets rid of an array copy that also got placed on top of the stack.

Convert an Array into a String

This one is way, way, waaaay beyond disgustingly elegant. And is in the same off-the-wall class as using **setdash** to instantly draw thousands of **perspective bricks**. Assume you have an array of positive integers in the 0 to 255 range. They might be an ASCII text message or they simply might be some data that you want to efficiently stash as a string. To convert…

```
/makestring {dup length string dup /NullEncode filter
3 -1 roll {1 index exch write} forall pop} def

[72 105 32 116 104 101 114 101]makestring ---> (Hi there!)
```

Huh? OK, here it is in English: Create a new string the length of the array. Make the string into a virtual **file** that can be written to. Then stuff the array integers into the string one by one.

Sort an Array of Strings Alphabetically

PostScript's **gt** and **lt** are even more powerful than you would first suspect when they are applied to strings. For they will automatically go through each entire string **character by character** and put them in **lexigraphic** (or "dictionary") order.

A plain old stack based **bubble sort** can be more than fast enough for an array of thousand or fewer strings…

```
/bubblesortalpha {mark exch aload pop counttomark /idx
exch store {0 1 idx 1 sub {pop 2 copy lt {exch} if idx 1 roll}
for idx 1 roll /idx idx 1 sub store idx 0 eq {exit} if} loop ]
} store

[(aardvark)(dragon)(badger)(camel)] bubblesortalpha ----->
[(aardvark)(badger)(camel)(dragon)]
```

One minor gotcha is that alphanumeric sorts will end up **case sensitive**. Your workarounds are capitalizing or trapping each first letter.

Sort an Array of Subarrays by Popularity

This gets real handy in a hurry for such things as **Weblog Reports**. Say you have an array of subarrays. Assume further that each of your subarrays will be of a form

[{filename} popularity_count] . You can modify our previous bubble sort to sort on the **popularity_count** integer subarray position…

```
/bubblesortpopular {mark exch aload pop counttomark /idx
exch store {0 1 idx 1 sub {pop 2 copy 1 get exch 1 get
gt {exch} if idx 1 roll} for idx 1 roll /idx idx 1 sub
store idx 0 eq {exit} if} loop ] } store

[ [(file1) 16][(file2) 5][(file3) 45][(file4) 2][(file5) 7] ]
bubblesortpopular ----->
[ [(file3) 45][(file1)16]][(file5) 7][(file2) 5][(file4) 2] ]
```

Any position of longer subarrays could be sorted numerically or alphanumerically simply by changing the position value of each **get**.

These routines seem more than fast enough. Being well under a second for a thousand or less array elements. Bubble sorts increase in time as **n squared**. **Other Sorts** may be faster for very large arrays.

Convert a Dictionary to an Array

PostScript dictionaries normally relate key-value pairs. While it is quite common for a key to be a **name** type, there is emphatically no such restriction in the **PostScript** language. **A PostScript dictionary can relate ANY two data types!** Something that seems to be one of the most incredibly powerful opportunities anywhere ever in all of computerdom. And one that remains largely unexplored.

Thus, **any** entry in a dictionary is simply "match the first to get the second". Unlike arrays, dictionaries automatically change their length as new items are added. Better yet, dictionaries offer the **known** command that greatly simplifies modifying an existing entry versus entering a new one. Thus, for certain paired data structures, **it may end up faster and simpler to start off with a dictionary instead of an array**.

Let's look at a specific example that recently has proven most handy. Say you want to extract individual scripts for all the log visitors to your website. This could end up as an array of two element subarrays. One per visitor. The subarrays could end up of form **[(23.156.67.121) [23 24 27 33 45]]** In this case, your visitor url **23.156.67.121** made hits on lines **23**, **24**, **27**, **33**, and **45** of your web log.

To proceed, start with a new dictionary. As you go through your log file, either **start** a new url definition or **add** to the line hits on an old one. The url can be either in the form of a conventional name or can remain a string. The self expanding feature of a dictionary combined with its **known** operator makes data creation fast and simple.

After you have read your entire web log, you can convert from a dictionary to an array of two element arrays…

```
/newarray mark olddictionary { mark 3 1 roll ] } forall ] store
```

Note that the **forall** operator behaves differently on a dictionary in that it returns **both** values of the key-value pair for each entry. See **ANALOGEB.PDF** for detailed example code.

Using PostScript to Write PostScript

All sorts of "leap tall buildings in a single bound" possibilities arise whenever you let your **PostScript** code write its own **new** and **deferred** executable procs. We have already seen how you can create a string and then do a **cvx** followed by a later **(command_string) exec** when actual execution is desirable.

Arrays can also let you build up future **PostScript** executables. And often may end up more powerful and convenient than strings. The only little trick is...

NEVER put anything into an array being stack built that you do not want to have execute immediately.

Instead, place the NAME OBJECT of any to-be-deferred proc on the stack. Followed by a cvx.

Let's look at an example. In PDF text layout, for an **autotracking url hot link**, you'll want your bounding box to move around as your text does. But the exact position won't be known until **after** you have made and executed your text justification routines. Ferinstance, your hotlink bounding box will probably stretch and have to move somewhat to the right during a fill justification.

To solve this dilemma, you make **two** passes through each text line. The first pass determines which strings are to be shown where and how. These substrings can be cataloged into an array. Then your justification calcs are made. Finally, each portion of the line is imaged when and where needed in its correct font size.

Here's how you save two crucial **but still unknown** values for a later autotracking url box...

```
    mark 0.33 /setgray cvx /currentpoint cvx /urly /exch
    cvx /store cvx /urlx /exch cvx /store cvx] cvx
```

which safely becomes...

```
    {0.33 setgray currentpoint /urly exch store /urlx exch
    store}
```

Note that **currentpoint** executes **later on** when you create your actual bounding box, and **not** at the time you build your array. Many more details on this example appear in **AUTOURL.PDF**.

With these sneaky deferral techniques, your **PostScript** can even write **JavaScript** code! Or anything you want in **any** other language.

For More Help

Additional **PostScript** and **Acrobat** and assistance is available per the previously shown web links. Custom programming and design services are now available at our standard consulting rates. Per our **InfoPack Services**. Or you can directly **email** me.

Additional **GuruGrams** columns await your ongoing support as a **Synergetics Partner**.