Dealing with arrays of text

Ben Klemens

30 August 2013

Dealing with text in C is annoying for several reasons, primary among them the fact that changing a string often involves a memory reallocation. In *21st Century C*, I talk about the lovely `asprintf` function; this was one of the most popular bits of the book; e.g., see the summary on this blog[1].

Here, I consider the next step: an array of text, which we get when we have text data in a data set. This entry will describe where I wound up in trying to design a setup for dealing with grids of text data.

Last time (entry #166), I talked about the straightforward `gsl_vector` and `gsl_matrix` functions, and the interface functions in Apophenia that smooth over some of the details. Similarly, the `text` element of the `apop_data` set is in no way special—it is declared as `char ***text`—but with the right interface functions, we can use this unremarkable structure reasonably gracefully.

- Declare with `apop_text_alloc`. This generates the pointer-to-pointer-to-pointer grid for you, which is already saving a lot of pain. This is really a realloc function, so if you change the size to larger or smaller, there's no memory leak. You can only produce a rectangular array of elements, though having a long list of pointers-to-`NULL` [As Apop. v1 will have] is rarely a serious problem.
- Add text with `apop_text_add`. This is really a realloc function, so if there's already text in that slot, there's no memory leak.
- Add lots of static text at once with `apop_text_fill`.
- Free the whole thing with `apop_text_free`, which you'll never use anyway, because you'll use `apop_data_free` to free the entire data set at once.

OK, we're done with memory management of strings in C: just use those three functions, and it's all taken care of. Again, other systems do it in other ways, such as generating a struct that includes string metadata, thus forcing users to always use the accessor methods of the object.

Accessing the elements of a 2D grid of structs is also straightforward, once you have the rules down. Given a data set declared via `apop_data *td = apop_text_alloc(NULL, 3, 3)`:

- `td->textsize[0]` is the row count; `td->textsize[1]` is the column count.

[1] `http://www.power-quant.com/?q=node/80`

- Element (i, j) is `td->text[i][j]`.

With the matrices and vectors, I was concerned with having a setup that gracefully dealt with matrices, vectors, and scalars. Why should I always have to specify two indices when I sometimes only need one or zero? We can do a similar thing using the rule that `*x` and `x[0]` are equivalent in C—you can think of one as syntactic sugar for the other [see §6.5.2.1(2) of the C standard].

- The oft-used row count is `*td->textsize`.
- If you know you have only a single column of text, get item i via `*td->text[i]`. If you know you have a single text element, it is `**td->text`.

So there's the string-handling system via the `apop_data` struct. Write using the allocate and add functions, free with the free function, and read directly from the `text` and `textsize` elements.

Some bonus functions Transpose via `apop_data_transpose`, which transposes both the matrix and text elements of the `apop_data` set that gets input. This is especially useful when you have a function that needs a list of strings, because the column of strings above was really a list of pointers-to-one-string, but each row of the text array is one pointer-to-strings as desired.

It might be nice to have a few view functions that view columns or subgrids. One could implement them the way that vector views were implemented in the GSL, by setting up a list of starting points and changing the `textsizes` accordingly. [That's a hint, for anybody who's interested in contributing. It's nontrivial, though.]

Use `apop_text_paste` to merge a grid of strings into a single string. This is an emulation of R's `paste` function, modified to more naturally handle a 2-D grid of text. With creative separators, you can produce all sorts of things with this. E.g., the documentation for the function prints a text grid as an HTML table, by putting the appropriate HTML tags between elements, between rows, and at the head and tail.

Conversely, if you need to split a string into parts, use `apop_regex`. Using parens in a regular expression tells the regex parser to return those parts of the regex to the user. The POSIX-standard regex parser will return a pointer to the end of the matched string, and we can pick up where that leaves off to parse again. The output from `apop_regex` will have columns of text corresponding to the parens in your regex, and columns corresponding to repeated application.

Here's an example that takes in a regex and a string, then parses it down and prints the results in a table. This would be pulling teeth in raw C, but with the right interface functions writing this in C is much like writing it in a pointer-free scripting language.

I wrote this as a shell script. Cut and paste it to your command line to have gcc produce `a.out` and run it as per the examples given.

```
gcc -xc - `pkg-config --libs --cflags apophenia` -g -Wall --std=gnu99 -O3 << "
    ----"
#include <apop.h>

int main(int argc, char **argv){
    Apop_stopif(argc == 2, return 1, 0, "Give two arguments: a regex and a string.");
    apop_data *txt;
    int returnval = apop_regex(argv[2], argv[1], &txt);
    Apop_stopif(!returnval, return 1, 0, "No matches found.");
    Apop_stopif(returnval==-1, return 2, 0, "Bad regex.");
    printf("%s", apop_text_paste(txt, .before=" [", .between_cols="] [", .between="]\n [", .
        after="]\n"));
}
----

a.out "([[:alpha:]]+), *([-[:digit:].]+)" "{pants, 23} {plants,-14} {dirigibles, 12.81}"
echo -----
a.out "((<[^<]*>)([^<]*)</[^>]*>)" "<b>pants</b> <em>blazers</em>"
echo -----
a.out "([[:alnum:].])" " hello. "
```