Hints & Tricks

Glisterings

Peter Wilson

> 'Tis better to be lowly born
> And range with humble livers in content
> Than to be perked up in a glist'ring grief
> And wear a golden sorrow.
>
> *Henry VIII*, William Shakespeare

The aim of this column is to provide odd hints or small pieces of code that might help in solving a problem or two while hopefully not making things worse through any errors of mine.

Corrections, suggestions, and contributions will always be welcome.

> To no one but the Son of Heaven does it belong to order ceremonies, to fix the measures, and to determine the written characters.
>
> *The Analects*, Confucius

1 Stringing along

In an earlier column [3] I mentioned that I might continue looking at character strings. Here is some basic code that can be used for examining each character in a simple string:

```
\catcode`\^^G=12
\newcommand*{\allchars}[1]{%
  \def\stuff{#1}\ifx\stuff\@empty\else
  \@llchars#1^^G\fi}
\def\@llchars#1#2^^G{%
  \def\letter{#1}\def\others{#2}%
  \ifx\letter\@empty\let\next\@gobble
  \else
    \doachar{#1}%
    \ifx\others\@empty \let\next\@gobble
    \else \let\next\@llchars \fi
  \fi
  \next#2^^G}
\catcode`\^^G=15
```

Here I have used the special character `^^G` as a marker for the end of the string. This is normally an invalid character but I temporarily changed its catcode to make it an 'other' character (like `@` normally is). The `\@gobble` macro is part of the LaTeX kernel; it takes one argument and does nothing with it. Buried inside the code `\allchars` calls a macro `\doachar{`⟨*char*⟩`}` for each character in the string. With this definition

```
\newcommand*{\doachar}[1]{\textit{#1}}
```

some examples of `\allchars` are:
`\allchars{allchars} ->` *allchars*
`\allchars{{\oe}rstead's} ->` *œrstead's*
`\allchars{} ->`
`\allchars{with spaces} ->` *withspaces*
The special case of an empty argument is handled in the `\allchars` macro itself, while everything else is dealt with by `\@llchars`. This keeps calling itself, grabbing one character from the initial string each time until all are used up, via a process called *tail recursion*, meaning that the last thing that it does is call itself (or effectively do nothing if all the characters have been processed).

Remember that with LaTeX, if you put any code that includes macros with `@` in their names it either has to go in a package file (a `.sty` file) or be surrounded by the `\makeatletter` and `\makeatother` pair of commands.

One unfortunate property of `\allchars` is that it discards all spaces in the original string. Spaces can be handled by a two-part process. The first part goes through the string word by word, where a word is a set of characters followed by a space. The second part then goes through each word character by character.

First some preliminaries and the main user command `\Upeach`.

```
\newif\if@newword
\def\checkrelax{\relax}
\catcode`\^^G=12
\newcommand*{\Upeach}[1]{%
  \@upeach#1^^G}
\def\@upeach#1^^G{%
  \def\stuff{#1 }%
  \expandafter\getaword\stuff ^^G}
```

The `\getaword` macro extracts the next word from the string (note the argument delimited by a space). It then calls `\getachar` with the word as its argument.

```
\long\def\getaword#1 {%
  \@newwordtrue
  \expandafter\getachar#1\relax}
```

`\getachar` gets the next 'letter' in the word. If it is `^^G` then the string is finished. If the letter is the same as `\relax` then it is a space and the macro must call `\getaword` to repeat the cycle. Otherwise it has a letter, calls `\doUpeach` to do something with it, and calls itself again to get the following letter.

```
\def\getachar#1{%
  \def\letter{#1}%
  \if\letter^^G\let\next\relax
  \else
    \ifx\letter\checkrelax
      \let\next\getaword
```

```
    \else
      \doUpeach{#1}%
      \let\next\getachar
    \fi
  \fi
  \next}
\catcode`\^^G=15
```

`\getachar` is another example of tail recursion.

The macro `\doUpeach` checks if a new word has just started. If so, it converts its argument into italic uppercase and sets `\@newwordfalse`. If its argument is not the first letter in a word it typesets it in a bold font. Of course this is not a realistic thing to do — it's merely to demonstrate that all the characters in the string have been examined.

```
\newcommand*{\doUpeach}[1]{%
  \if@newword
    \space\textit{\MakeUppercase{#1}}%
    \@newwordfalse
  \else \textbf{#1}\fi}
```

Here are a couple of examples:

```
\Upeach{string with spaces} ->
```
***S*tring *W*ith *S*paces**

```
\Upeach{{\oe}rstead's rule} ->
```
***Œ*rstead's *R*ule**

These macros work for simple strings but are likely to fail if there are accents or anything else to disturb the even tenor of simple characters. The earlier column [3] gave an indication of how such problems might be resolved. On the other hand, it could be a lot simpler and quicker to change the strings by hand using your normal text editor.

> Here we go loop de loop.
> Here we go loop de li.
> Here we go loop de loop
> On a Saturday night.
>
> *Loop de loop*, Johnny Thunder?

2 Loops

There are occasions when you need to perform a repetitive action that does not involve string processing. TEX provides a `\loop ... \repeat` which can be useful in some circumstances. The general scheme is like this:

```
\loop
  <lots of useful commands>
\if<some condition>
  <more code>
\repeat
```

TEX processes the commands following the `\loop` and then performs the `\if` test (without any closing `\fi`). If the test is true TEX will then process the `<more code>` and start again with the first batch of commands. If the condition is false it will do whatever comes after the `\repeat`.

LATEX, among other internal facilities, provides a mechanism for going through a list of things that are separated by commas (like the option list for a class or package). This scheme looks like:

```
\@for\scratch:=<list>\do{%
  <something with \scratch>}
```

where `\scratch` is some command name and `<list>` is a comma-separated list. It takes each element of the list in turn, defines `\scratch` as that element and then does whatever you tell it to do with it. This continues until the list is exhausted.

It is easier to see how these work with a real example. The following is a very stripped down version of some code from the memoir class [2]. It provides a means of putting a list of things into a tabular form without having to worry about signifying the end of each row. The command is:
`\fillrows{`⟨*width*⟩`}{`⟨*numcols*⟩`}{`⟨*comma separated list*⟩`}`
which will create a centered tabular form of overall width ⟨*width*⟩ and ⟨*numcols*⟩ columns, with the elements from ⟨*comma separated list*⟩ filling up the tabular row by row (i.e., left to right, top to bottom). I got the initial idea from *TEX for the Impatient* [1] which gave a TEX version, filling top to bottom and left to right.

First some counters and lengths, etc., that we need. Be warned, much of the code below you won't want to know about and I'm not going to try and explain it. In LATEX this is the kind of stuff that is hidden within the `tabular` environment.

```
\newcount\CT@cols      % number of cols
\newcount\@cellstogo   % columns left
\newdimen\CT@col@width % column width
\newtoks\crtok
  \crtok = {\cr}%
```

Now we can start on `\fillrows` itself, which takes three arguments — the overall width, the number of columns, and the list of entries. The first part sets up counters based on the number of columns.

```
\newcommand{\fillrows}[3]{\par\begingroup
  \CT@cols=#2\relax
  \@cellstogo=\CT@cols
```

The next bit defines code that will be called after each entry is put into the tabular; it will insert either a `&` or the internal form of `\\`.

```
  \def\@endcolactions{%
    \global\advance\@cellstogo\m@ne
    \ifnum\@cellstogo<\@ne
      \global\@cellstogo=\CT@cols
      \the\crtok
```

```
  \else
    &
  \fi}%
```

Calculate the column widths and start off the tabular by defining the preamble (the general layout of the tabular).

```
\CT@col@width=#1
\divide\CT@col@width \CT@cols
\penalty 10000\relax
\noindent
\vskip -\z@
\def\@preamble{}%
\begingroup
  \let\@sharp\relax
```

Now comes a `\loop...\repeat` going over all but one of the columns, and for each column extending the `\@preamble` by adding some spacing and a `&`.

```
  \ifnum\CT@cols>\@ne
    \loop
      \g@addto@macro{\@preamble}{%
        \hb@xt@ \CT@col@width
          {\strut\relax\@sharp\hfil} &}%
      \advance\CT@cols\m@ne
    \ifnum\CT@cols>\@ne
    \repeat
  \fi
```

The `&` is not required for the last column.

```
  \g@addto@macro{\@preamble}{%
    \hb@xt@ \CT@col@width
      {\strut\relax\@sharp\hfil}}%
\endgroup
```

(The above code sets each column to a fixed width (`\CT@col@width`). Commenting out the two lines that start with `\hb@xt@` will result in each column being set to its natural width, just wide enough for the widest entry in the column.) Now finish up the preliminaries.

```
\let\@sharp ##
\tabskip\fill
\halign to\hsize \bgroup
  \tabskip\z@
  \@preamble
  \tabskip\fill\cr
```

The entries are added to the tabular, using a `\@for` loop to extract each entry from the comma-separated list.

```
  \@for\@tempa:=#3\do{%
    \@tempa\unskip\space\@endcolactions}%
```

All the entries have been dealt with, so wrap everything up.

```
  \the\crtok \egroup \endgroup \par}
```

As a simple example, the code below creates the following tabular:

```
\fillrows{0.7\textwidth}{3}{ one, two,
        three, four, five, six, seven}
```

one	two	three
four	five	six
seven		

And here is the result of another `\fillrows`, this time with five columns set to their natural width.

That's	all	folks!	Until	we
meet	again	...		

References

[1] Paul W. Abrahams, Karl Berry, and Kathryn A. Hargreaves. *TEX for the Impatient*. Addison-Wesley, 1990. (Available on CTAN in `info/impatient`).

[2] Peter Wilson. The memoir class for configurable typesetting, 2004. (Available on CTAN in `latex/macros/contrib/memoir`).

[3] Peter Wilson. Glisterings. *TUGboat*, 26(3):253–255, 2005.

◇ Peter Wilson
18912 8th Ave. SW
Normandy Park, WA 98166
USA
herries dot press (at)
earthlink dot net