Glisterings

Peter Wilson

The raging waves doth belching upwardcast
The wretched wrackes that round about doe fleete,
The silken sayles and glistering golden Mast,
Lies all to torne and trodden under feete.

The Ship of safegarde, Barnabe Googe

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.

The main topic this time is macro definition. Questions about this, particularly with respect to LaTeX, are fairly regular on the `comp.text.tex` newsgroup. But first…

The lines are fallen unto me in pleasant places; yea I have a goodly heritage.

Psalm 16, verse 6

1 More on paragraphs

Donald Knuth sent me a version of the following code saying

> 'I've found this macro to be useful for checking out a `\parshape` specification before cluttering it up with actual text.'

```
% parshape.tex, featuring a possibly useful
%    macro by Don Knuth, April 2007
% \parshapetest{n} will typeset n lines of
%    horizontal rules using the current
%    paragraph shape (as specified by
%    \hangindent, \hangafter, \parshape, or
%    none of the above)
\def\parshapetest#1{%
  \leavevmode%% DEK originally had \indent here
  \count255=1 \loop
    \ifnum\count255<#1
    \null\leaders\hrule\hfil\null\break
    \advance\count255 by 1   \repeat
  \null\leaders\hrule\hfil\hskip-\parfillskip
  \null\par}
```

Unfortunately it was too late to incorporate it into the last column [8] which was about how to typeset variously shaped paragraphs. It was doubly unfortunate because when I tried using `\parshapetest` on some of the examples I found that I had misunderstood some aspects of paragraph setting.

`\parshapetest{`⟨*num*⟩`}` draws ⟨*num*⟩ lines according to the current paragraph shape specification, which doesn't sound very exciting but can save a lot of fiddling trying to get the right number of words for a more realistic trial layout.

For instance, I tried this example from [8]

```
\begingroup
\hangindent=3pc \hangafter=-2
\parshapetest{4}
\endgroup
```

which, to my surprise, resulted in:

What I hadn't realised was that even with specifying `\hangindent` and `\hangafter`, `\parindent` was applied to the first line of the shaped paragraph. The effect that I had expected is obtained as below.

```
\begingroup
\parindent=0pt
\hangindent=3pc \hangafter=-2
\parshapetest{4}
\endgroup
```

which results in:

Following this I tried the `\hangfrom` example from the same column which demonstrated a hanging paragraph. The macro was defined as:

```
\newcommand*{\hangfrom}[1]{%
  \setbox\@tempboxa\hbox{{#1}}%
  \hangindent \wd\@tempboxa
  \noindent\box\@tempboxa}
```

And a demonstration is:

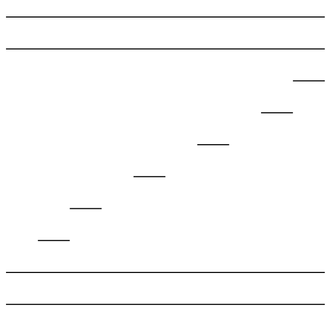
```
\hangfrom{$\Rightarrow$\space}
\parshapetest{3}
```

⇒

Here's a more interesting paragraph shape, and the result of testing it:

```
\newdimen\zide
  \zide=\baselineskip
\newcommand*{\aparshape}{%
\parshape=10 0pt 10\zide % 1
             0pt 10\zide % 2
             9\zide \zide % 3
             8\zide \zide % 4
             6\zide \zide % 5
             4\zide \zide % 6
             2\zide \zide % 7
```

```
              \zide \zide % 8
              0pt 10\zide % 9
              0pt 10\zide % 10
}
\aparshape
\noindent\parshapetest{10}
```



Try using this paragraph shape with a text of 76 ‘z’ characters with spaces between each, like this:

```
\aparshape
\noindent
z z z z z z z z z z z z z z z z z z z z
etc
```

Replying to a request on the `comp.text.tex` newsgroup by Stephen Moye, Paul Vojta [7] posted the following code[1] for setting the first line of a paragraph flushleft, the next centered and the final line flushright.

```
\newcommand*{\leftcenterright}{%
  \leftskip=0pt plus 1fil
  \rightskip=0pt plus 1fil
  \parfillskip=0pt plus -1fil
  \parindent=0pt
  \everypar={\hskip0pt plus -1fil}}
```

This should either be used in a group, or you can use the following macro to return to the regular paragraph style, where you have previously specified `\myparindent` as the normal value of `\parindent`.

```
\newcommand*{\regularpar}{%
  \leftskip=0pt plus 0pt minus 0pt
  \rightskip=\leftskip
  \parfillskip=0pt plus 1fil
  \parindent=\myparindent
  \everypar{}}
```

Following is an example of a `\leftcenterright` paragraph, typeset from:

```
\leftcenterright
First line \\ Second line \break
Third line \break Last line \par
\regularpar
```

First line
Second line
Third line
Last line

[1] For convenience I have put Paul’s code into a macro.

Who will change old lamps for new? ...new lamps for old ones?

Arabian Nights: The History of Aladdin

2 LaTeX’s defining triumvirate

The macro provided by LaTeX for defining new commands is somewhat simpler than the TeX macro upon which it is based. This is the LaTeX one:
`\newcommand{`⟨*cmd*⟩`}[`⟨*num*⟩`][`⟨*arg1*⟩`]{`⟨*defn*⟩`}`
where ⟨*cmd*⟩ is the name, including the backslash (e.g., `\amacro`), of the new macro being defined and ⟨*defn*⟩ is the definition of the new macro, which may be simply some text to be typeset or something very complex. The optional ⟨*num*⟩ argument specifies the number of arguments that the new macro will take; if given this must be at least one and at most nine. The new macro will take an optional argument if ⟨*arg1*⟩ is given, where ⟨*arg1*⟩ is the default value of the first argument. The macro resulting from `\newcommand` is, in TeX terms, a *long* macro, meaning that an argument may consist of more than one paragraph or, equivalently, include a `\par`. There is also a star form of the command (`\newcommand*`) which creates a macro where paragraph(s) are not allowed in an argument to the new macro. If ⟨*cmd*⟩ has been defined previously LaTeX will give an error message.

The LaTeX macro
`\renewcommand{`⟨*cmd*⟩`}[`⟨*num*⟩`][`⟨*arg1*⟩`]{`⟨*defn*⟩`}`
and its companion `\renewcommand*`, are similar to `\newcommand` except that they change the definition of ⟨*cmd*⟩, which *must* have been defined earlier, otherwise LaTeX will complain.

The third member of LaTeX’s macro definition macros is:
`\providecommand{`⟨*cmd*⟩`}[`⟨*num*⟩`][`⟨*arg1*⟩`]{`⟨*defn*⟩`}`
which acts like `\newcommand` if ⟨*cmd*⟩ has not been defined, otherwise it silently does nothing. Again, there is a starred version of the command.

If you want to make sure that your definition for ⟨*cmd*⟩ is used regardless of whether or not it has been defined before you can do this:

```
% ensure \amacro is defined
\providecommand{\amacro}{}
% change the definition
\renewcommand{\amacro}...
```

Within the ⟨*defn*⟩ argument to the macros the use of the first argument, if any, is denoted by `#1`, the second by `#2`, and so on up to the ninth which is denoted by `#9`. The arguments can be used as many times as needed and in any order.

A question that pops up now and then on the `comp.text.tex` newsgroup is how to define a macro that takes more than nine arguments. The answer is to split it up into two or more macros each of which handles a portion of the required number. For, say, 11 arguments:

```
\newcommand{\xiargs}[9]{%
  % 9 args used here then
  \xtrargs}
\newcommand{\xtrargs}[2]{%
  % use last 2 args here
  % #1 and #2 are the apparent 10th & 11th args
}
```

The user calls `\xiargs` with the 11 arguments, and `\xiargs` processes the first 9 of these. It then calls `\xtrargs`, which is effectively hidden from view, to process the remaining 2 arguments. If you need to use, say, the 4th argument within `\xtrargs` this can be easily accomplished:

```
\newcommand{\xiargs}[9]{%
  % 9 args used here then
  \xtrargs{#4}}
\newcommand{\xtrargs}[3]{%
  % #1 here is #4 from \xiargs and
  % #2 and #3 are the apparent 10th & 11th args
}
```

As a lead in to the next section, here is another way of getting the 4th argument into `\xtrargs`:

```
\newcommand{\xiargs}[9]{%
  % 9 args used here including
  \def\ivarg{#4}%
  % then
  \xtrargs}
\newcommand{\xtrargs}[2]{%
  % #1 and #2 are the apparent 10th & 11th args
  % call \ivarg for original 4th arg
}
```

where `\def` is the TeX command for defining a command.

This kind of code can obviously be extended to handle as many arguments as you wish, but after a while it might be easier to use the keyval package [3], or the later extension called xkeyval [2], which provide a very different approach. You name each argument and the user can use as many or as few of these as he deems necessary.

He who can properly define and divide is to be considered a god.

Novum Organum, Francis Bacon quoting Plato

3 TeX's dictator

TeX has an all-purpose command for defining new macros, namely `\def`. There are too many aspects to this to cover them all in a short article; Knuth [5, ch. 20] provides the definitive explanation, but you may find that Eijkhout [4, ch. 11] or Abrahams *et al.* [1, chs. 4 and 9] are more accessible or helpful.

The syntax of the `\def` command is unlike anything you see in an author's view of LaTeX.

`\def`⟨*cmd*⟩⟨*paramspec*⟩{⟨*defn*⟩}

As in the LaTeX formulation, ⟨*cmd*⟩ is the name, including the backslash (e.g., `\amacro`), of the new macro being defined and ⟨*defn*⟩ is the definition of the new macro, just as with LaTeX. Note that there are no braces around ⟨*cmd*⟩.

The ⟨*paramspec*⟩ is where you specify the appearance of any arguments to ⟨*cmd*⟩. Each argument is denoted by `#1`, `#2`, etc., in ⟨*paramspec*⟩; these must be in numerical order, and spaces within ⟨*paramspec*⟩ are significant. Below are two equivalent pieces of (LA)TeX code:

```
\newcommand*{\amacro}[2]{....} % LaTeX
\def\amacro#1#2{....}          % TeX
... \amacro{foo}{bar} ...      % (La)TeX
```

That finishes the simple bit, except to say that if you need an argument to consist of one or more paragraphs, by including a blank line or a `\par`, then the macro must be *long*. Also TeX gives no warning if you `\def` a macro that has already been defined — it just throws the old definition away. Be careful of this as it is not a good idea to inadvertently redefine some vital macro that you did not know existed. Anyway, here are two more equivalent pieces of code:

```
\renewcommand{\amacro}[2]{....}       % LaTeX
\long\def\amacro#1#2{....}            % TeX
... \amacro{A paragraph\par}{bar} ... % (La)TeX
```

When the ⟨*paramspec*⟩ consists only of parameters (the `#1` etc.) they are said to be *undelimited*; simplistically these correspond to LaTeX's mandatory arguments. On the other hand, if any non-parameter tokens (that is, anything except a `#n` or the opening `{` of the {⟨*defn*⟩}) occur after a `#n` then that parameter is said to be *delimited*. When the new macro is called, the argument for a delimited parameter does not end until TeX encounters the delimiting character(s). Internally, LaTeX uses delimited parameters to implement optional arguments.

Suppose we need a macro that looks like this:

`\where{foo}(x,y)`

where `foo`, `x` and `y` are the arguments to `\where`. The LaTeX commands described above can't handle this, but TeX can:

```
\def\where#1(#2,#3){#1 in #2 #3}
```

and calling

```
\textit{%
\where{A nightingale sang}(Berkely,Square)}
```

results in

A nightingale sang in Berkely Square

Perhaps you need a command that comes in two versions, as `\newcommand` does. The LaTeX kernel includes a macro called `\@ifnextchar`, whose syntax is like this:
`\@ifnextchar`⟨*char*⟩{⟨*yes*⟩}{⟨*no*⟩}
It looks to see if the next non-space character in the input text is ⟨*char*⟩. If it is it executes the ⟨*yes*⟩ argument, otherwise it executes the ⟨*no*⟩ argument. The kernel also provides the next command:
`\@ifstar`{⟨*yes*⟩}{⟨*no*⟩}
which looks to see if the next character is a `*` and if it is it gobbles up the `*` and executes the ⟨*yes*⟩ argument, otherwise it executes the ⟨*no*⟩ argument. It is defined as follows:

```
\long\def\@firstoftwo#1#2{#1}
\def\@ifstar#1{%
  \@ifnextchar *{\@firstoftwo{#1}}}
```

Now you can define your own (un)starred command pair, like this:

```
\makeatletter  % if not in a .cls or .sty file
\def\maybestar{%
  \@ifstar{\@maybestarS}{\@maybestar}}
  % handle starred version
\def\@maybestarS#1#2{Star (#1) and (#2).}
  % handle plain version
\def\@maybestar#1#2{(#1) and (#2).}
\makeatother   % if not in a .cls or .sty file
```

The end result is a macro with a starred and unstarred version that takes two arguments. A pair of example results are:
`\maybestar*{1st}{2nd}` → Star (1st) and (2nd).
`\maybestar{1st}{2nd}` → (1st) and (2nd).

If you would like to use another character, say a `?`, in place of the `*`, here's a way of doing it.

```
\def\maybeQ{%
  \@ifnextchar ?{\@maybeQ}{\@maybe}}
\def\@maybeQ#1#2#3{Query (#2) and (#3).}
\def\@maybe#1#2{(#1) and (#2).}
```

Unlike the starring code where `\@ifstar` got rid of the `*` the `\@maybeQ` macro has to discard the `?` which is the first character it will see; TeX treats a single character[2] as an argument so `\@maybeQ` is defined such that it throws away its first argument.

A pair of example results are:
`\maybeQ?{1st}{2nd}` → Query (1st) and (2nd).
`\maybeQ{1st}{2nd}` → (1st) and (2nd).

Maybe you would like a LaTeX command that takes two optional arguments and one required one. Heiko Oberdiek has produced a comprehensive package for creating such macros [6] but as another TeX example here is a simple method that might be useful for the odd occasion. The result will be a LaTeX macro, `\twoopt`, that takes one required and two optional arguments. The defaults for the two optional arguments are to be 'opt1' and 'opt2', respectively and unimaginatively.

```
\def\twoopt{%
  \@ifnextchar [{\@twoopt}{\@twoopt[opt1]}}
\def\@twoopt[#1]{%
  \@ifnextchar [%
      {\@@twoopt{#1}}{\@@twoopt{#1}[opt2]}}
\def\@@twoopt#1[#2]#3{%
  1 (#1) 2 (#2) 3 (#3)}
```

Don't forget that this has to be defined when LaTeX thinks that `@` is a letter. Trying this out we get:
`\twoopt{no opts}` → 1 (opt1) 2 (opt2) 3 (no opts)
`\twoopt[foo]{one opt}` → 1 (foo) 2 (opt2) 3 (one opt)
`\twoopt[bar][baz]{two opts}` → 1 (bar) 2 (baz) 3 (two opts)

[2] More precisely, a token, but now is not the time to get into all that.

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] Hendri Adriaens. The xkeyval package, 2005. Available on CTAN in `latex/macros/contrib/xkeyval`.

[3] David Carlisle. The keyval package, 1999. Available on CTAN in `latex/macros/required/graphics`.

[4] Victor Eijkhout. *TeX by Topic, A TeXnician's Reference.* Addison-Wesley, 1991. ISBN 0-201-56882-9. Available at `http://www.eijkhout.net/tbt/`.

[5] Donald E. Knuth. *The TeXbook.* Addison-Wesley, 1984. ISBN 0–201–13448–9.

[6] Heiko Oberdiek. The twoopt package: Definitions with two optional arguments, 1999. Available on CTAN in `latex/macros/contrib/oberdiek`.

[7] Paul Vojta. Re: New York Times headline style. Post to `comp.text.tex` newsgroup, 10 July 2007.

[8] Peter Wilson. Glisterings. *TUGboat*, 28(2):229–232, 2007.

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