low level TeX

grouping

Contents

1 Introduction

This is a rather short explanation. I decided to write it after presenting the other topics at the 2019 ConTEXt meeting where there was a question about grouping.

1.1 Pascal

In a language like Pascal, the language that TEX has been written in, or Modula, its successor, there is no concept of grouping like in TEX. But we can find keywords that suggests this:

```
for i := 1 to 10 do begin ... end
```

This language probably inspired some of the syntax of TEX and MetaPost. For instance an assignment in MetaPost uses `:=` too. However, the `begin` and `end` don't really group but define a block of statements. You can have local variables in a procedure or function but the block is just a way to pack a sequence of statements.

1.2 TEX

In TEX macros (or source code) the following can occur:

```
\begingroup
    ...
\endgroup
```

as well as:

```
\bgroup
    ...
\egroup
```

Here we really group in the sense that assignments to variables inside a group are forgotten afterwards. All assignments are local to the group unless they are explicitly done global:

```
\scratchcounter=1
\def\foo{foo}
\begingroup
    \scratchcounter=2
    \global\globalscratchcounter=2
    \gdef\foo{FOO}
\endgroup
```

Here `\scratchcounter` is still one after the group is left but its global counterpart is now two. The `\foo` macro is also changed globally.

Although you can use both sets of commands to group, you cannot mix them, so this will trigger an error:

```
\bgroup
\endgroup
```

The bottomline is: if you want a value to persist after the group, you need to explicitly change its value globally. This makes a lot of sense in the perspective of TEX.

1.3 MetaPost

The MetaPost language also has a concept of grouping but in this case it's more like a programming language.

```
begingroup ;
    n := 123 ;
engroup ;
```

In this case the value of `n` is 123 after the group is left, unless you do this (for numerics there is actually no need to declare them):

```
begingroup ;
    save n ; numeric n ; n := 123 ;
engroup ;
```

Given the use of MetaPost (read: MetaFont) this makes a lot of sense: often you use macros to simplify code and you do want variables to change. Grouping in this language serves other purposes, like hiding what is between these commands and let the last expression become the result. In a `vardef` grouping is implicit.

So, in MetaPost all assignments are global, unless a variable is explicitly saved inside a group.

1.4 Lua

In Lua all assignments are global unless a variable is defines local:

```
local x = 1
local y = 1
for i = 1, 10 do
    local x = 2
    y = 2
end
```

Here the value of `x` after the loop is still one but `y` is now two. As in LuaTEX we mix TEX, MetaPost and Lua you can mix up these concepts. Another mixup is using `:=`, `endfor`, `fi` in Lua after done some MetaPost coding or using `end` instead of `endfor` in MetaPost which can make the library wait for more without triggering an error. Proper syntax highlighting in an editor clearly helps.

1.5 C

The Lua language is a mix between Pascal (which is one reason why I like it) and C.

```
int x = 1 ;
int y = 1 ;
for (i=1; i<=10;i++) {
    int x = 2 ;
    y = 2 ;
}
```

The semicolon is also used in Pascal but there it is a separator and not a statement end, while in MetaPost it does end a statement (expression).

2 Kinds of grouping

Explicit grouping is accomplished by the two grouping primitives:

```
\begingroup
    \sl render slanted here
\endgroup
```

However, often you will find this being used:

```
{\sl render slanted here}
```

This is not only more compact but also avoids the `\endgroup` gobbling following spaces when used inline. The next code is equivalent but also suffers from the gobbling:

```
\bgroup
    \sl render slanted here
\egroup
```

The `\bgroup` and `\egroup` commands are not primitives but aliases (made by `\let`) to the left and right curly brace. These two characters have so called category codes that signal that they can be used for grouping. The *can be* here suggest that there are other purposes and indeed there are, for instance in:

```
\toks 0 = {abs}
\hbox {def}
```

In the case of a token list assignment the curly braces fence the assignment, so scanning stops when a matching right brace is found. The following are all valid:

```
\toks 0 = {a{b}s}
\toks 0 = \bgroup a{b}s}
\toks 0 = {a{\bgroup b}s}
\toks 0 = {a{\egroup b}s}
\toks 0 = \bgroup a{\bgroup b}s}
\toks 0 = \bgroup a{\egroup b}s}
```

They have in common that the final fence has to be a right brace. That the first one can be a an alias is due to the fact that the scanner searches for a brace equivalent when it looks for the value. Because the equal is optional, there is some lookahead involved which involves expansion and possibly push back while once scanning for the content starts just tokens are collected, with a fast check for nested and final braces.

In the case of the box, all these specifications are valid:

```
\hbox {def}
\hbox \bgroup def\egroup
\hbox \bgroup def}
\hbox \bgroup d{e\egroup f}
\hbox {def\egroup
```

This is because now the braces and equivalent act as grouping symbols so as long as they match we're fine. There is a pitfall here: you cannot mix and match different grouping, so the next issues an error:

```
\bgroup xxx\endgroup   % error
\begingroup xxx\egroup % error
```

This can make it somewhat hard to write generic grouping macros without trickery that is not always obvious to the user. Fortunately it can be hidden in macros like the helper `\groupedcommand`. In LuaMetaTeX we have a clean way out of this dilemma:

```
\beginsimplegroup xxx\endsimplegroup
\beginsimplegroup xxx\endgroup
\beginsimplegroup xxx\egroup
```

When you start a group with `\beginsimplegroup` you can end it in the three ways shows above. This means that the user (or calling macro) doesn't take into account what kind of grouping was used to start with.

When we are in math mode things are different. First of all, grouping with `\begingroup` and `\endgroup` in some cases works as expected, but because the math input is converted in a list that gets processed later some settings can become persistent, like changes in style or family. You can bet better use `\beginmathgroup` and `\endmathgroup` as they restore some properties. We also just mention the `\frozen` prefix that can be used to freeze assignments to some math specific parameters inside a group.

3 Hooks

In addition to the original `\aftergroup` primitive we have some more hooks. They can best be demonstrated with an example:

```
\begingroup \bf
    %
    \aftergroup   A \aftergroup   1
    \atendofgroup B \atendofgroup 1
    %
    \aftergrouped   {A2}
    \atendofgrouped {B2}
    %
    test
\endgroup
```

These collectors are accumulative. Watch how the bold is applied to what we inject before the group ends.

test B1B2A1A2

4 Local versus global

When TEX enters a group and an assignment is made the current value is stored on the save stack, and at the end of the group the original value is restored. In LuaMetaTEX this mechanism is made a bit more efficient by avoiding redundant stack entries. This is also why the next feature can give unexpected results when not used wisely.

Now consider the following example:

```
\newdimension\MyDimension

\starttabulate[||||]
    \NC          \MyDimension10pt \the\MyDimension
    \NC \advance\MyDimension10pt \the\MyDimension
    \NC \advance\MyDimension10pt \the\MyDimension \NC \NR
    \NC          \MyDimension10pt \the\MyDimension
    \NC \advance\MyDimension10pt \the\MyDimension
    \NC \advance\MyDimension10pt \the\MyDimension \NC \NR
\stoptabulate
```

```
10.0pt   10.0pt   10.0pt
10.0pt   10.0pt   10.0pt
```

The reason why we get the same values is that cells are a group and therefore the value gets restored as we move on. We can use the `\global` prefix to get around this

```
\starttabulate[||||]
    \NC \global        \MyDimension10pt \the\MyDimension
    \NC \global\advance\MyDimension10pt \the\MyDimension
    \NC \global\advance\MyDimension10pt \the\MyDimension \NC \NR
    \NC \global        \MyDimension10pt \the\MyDimension
    \NC \global\advance\MyDimension10pt \the\MyDimension
    \NC \global\advance\MyDimension10pt \the\MyDimension \NC \NR
\stoptabulate
```

```
10.0pt   20.0pt   30.0pt
10.0pt   20.0pt   30.0pt
```

Instead of using a global assignment and increment we can also use the following

```
\constrained\MyDimension\zeropoint
\starttabulate[||||]
```

```
    \NC \retained        \MyDimension10pt \the\MyDimension
    \NC \retained\advance\MyDimension10pt \the\MyDimension
    \NC \retained\advance\MyDimension10pt \the\MyDimension \NC \NR
    \NC \retained        \MyDimension10pt \the\MyDimension
    \NC \retained\advance\MyDimension10pt \the\MyDimension
    \NC \retained\advance\MyDimension10pt \the\MyDimension \NC \NR
\stoptabulate
```

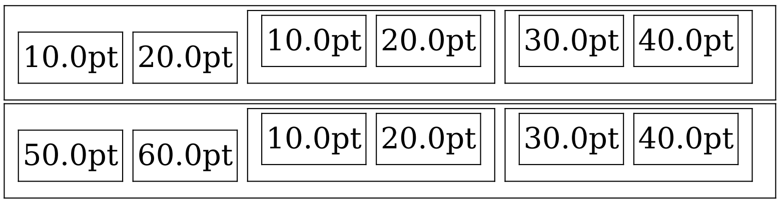
```
10.0pt  20.0pt  30.0pt
10.0pt  20.0pt  30.0pt
```

So what is the difference with the global approach? Say we have these two buffers:

```
\startbuffer[one]
    \global\MyDimension\zeropoint
    \framed {
        \framed {\global\advance\MyDimension10pt \the\MyDimension}
        \framed {\global\advance\MyDimension10pt \the\MyDimension}
    }
    \framed {
        \framed {\global\advance\MyDimension10pt \the\MyDimension}
        \framed {\global\advance\MyDimension10pt \the\MyDimension}
    }
\stopbuffer

\startbuffer[two]
    \global\MyDimension\zeropoint
    \framed {
        \framed {\global\advance\MyDimension10pt \the\MyDimension}
        \framed {\global\advance\MyDimension10pt \the\MyDimension}
        \getbuffer[one]
    }
    \framed {
        \framed {\global\advance\MyDimension10pt \the\MyDimension}
        \framed {\global\advance\MyDimension10pt \the\MyDimension}
        \getbuffer[one]
    }
\stopbuffer
```

Typesetting the second buffer gives us:

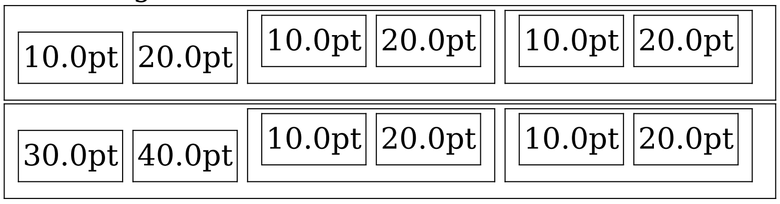


When we want to have these entities independent and not use different variables, the global settings bleeding from one into the other entity is messy. Therefore we can use this:

```
\startbuffer[one]
    \constrained\MyDimension\zeropoint
    \framed {
        \framed {\retained        \MyDimension10pt \the\MyDimension}
        \framed {\retained\advance\MyDimension10pt \the\MyDimension}
    }
    \framed {
        \framed {\retained        \MyDimension10pt \the\MyDimension}
        \framed {\retained\advance\MyDimension10pt \the\MyDimension}
    }
\stopbuffer

\startbuffer[two]
    \constrained\MyDimension\zeropoint
    \framed {
        \framed {\retained\advance\MyDimension10pt \the\MyDimension}
        \framed {\retained\advance\MyDimension10pt \the\MyDimension}
        \getbuffer[one]
    }
    \framed {
        \framed {\retained\advance\MyDimension10pt \the\MyDimension}
        \framed {\retained\advance\MyDimension10pt \the\MyDimension}
        \getbuffer[one]
    }
\stopbuffer
```

Now we get this:



The `\constrained` prefix makes sure that we have a stack entry, without being clever with respect to the current value. Then the `\retained` prefix can do its work reliably and avoid pushing the old value on the stack. Without the constrain it gets a bit unpredictable because then it all depends on where further up the chain the value was put on the stack. Of course one can argue that we should not have the “save stack redundant entries optimization” but that's not going to be removed.

4 Colofon

Author	Hans Hagen
ConTEXt	2023.08.04 01:06
LuaMetaTEX	211.0
Support	www.pragma-ade.com
	contextgarden.net