low level TEX

scope

Contents

1 Introduction

When I visited the file where register allocations are implemented I wondered to what extend it made sense to limit allocation to global instances only. This chapter deals with this phenomena.

2 Registers

In TEX definitions can be local or global. Most assignments are local within a group. Registers and definitions can be assigned global by using the `\global` prefix. There are also some properties that are global by design, like `\prevdepth`. A mixed breed are boxes. When you tweak its dimensions you actually tweak the current box, which can be an outer level. Compare:

```
\scratchcounter = 1
here the counter has value 1
\begingroup
    \scratchcounter = 2
    here the counter has value 2
\endgroup
here the counter has value 1
```

with:

```
\setbox\scratchbox=\hbox{}
here the box has zero width
\begingroup
    \wd\scratchbox=10pt
    here the box is 10pt wide
\endgroup
here the box is 10pt wide
```

It all makes sense so a remark like “Assignments to box dimensions are always global” are sort of confusing. Just look at this:

```
\setbox\scratchbox=\hbox to 20pt{}
here the box is \the\wd\scratchbox\ wide\par
\begingroup
    \setbox\scratchbox=\hbox{}
    here the box is \the\wd\scratchbox\ wide\par
    \begingroup
        \wd\scratchbox=15pt
        here the box is \the\wd\scratchbox\ wide\par
    \endgroup
    here the box is \the\wd\scratchbox\ wide\par
\endgroup
here the box is \the\wd\scratchbox\ wide\par
```

here the box is 20.0pt wide
here the box is 0.0pt wide
here the box is 15.0pt wide
here the box is 15.0pt wide
here the box is 20.0pt wide

If you don't think about it, what happens is what you expect. Now watch the next variant:

The `\global` is only effective for the current box. It is good to realize that when we talk registers, the box register behaves just like any other register but the manipulations happen to the current one.

```
\setbox\scratchbox=\hbox to 20pt{}
here the box is \the\wd\scratchbox\ wide\par
\begingroup
    \setbox\scratchbox=\hbox{}
    here the box is \the\wd\scratchbox\ wide\par
    \begingroup
        \global\wd\scratchbox=15pt
        here the box is \the\wd\scratchbox\ wide\par
    \endgroup
    here the box is \the\wd\scratchbox\ wide\par
\endgroup
here the box is \the\wd\scratchbox\ wide\par
```

here the box is 20.0pt wide

here the box is 0.0pt wide
here the box is 15.0pt wide
here the box is 15.0pt wide
here the box is 20.0pt wide

```
\scratchdimen=20pt
here the dimension is \the\scratchdimen\par
\begingroup
    \scratchdimen=0pt
    here the dimension is \the\scratchdimen\par
    \begingroup
        \global\scratchdimen=15pt
        here the dimension is \the\scratchdimen\par
    \endgroup
    here the dimension is \the\scratchdimen\par
\endgroup
here the dimension is \the\scratchdimen\par
```

here the dimension is 20.0pt
here the dimension is 0.0pt
here the dimension is 15.0pt
here the dimension is 15.0pt
here the dimension is 15.0pt

3 Allocation

The plain TEX format has set some standards and one of them is that registers are allocated with `\new...` commands. So we can say:

```
\newcount\mycounta
\newdimen\mydimena
```

These commands take a register from the pool and relate the given name to that entry. In ConTEXt we have a bunch of predefined scratch registers for general use, like:

```
scratchcounter    : \meaningfull\scratchcounter
scratchcounterone : \meaningfull\scratchcounterone
scratchcountertwo : \meaningfull\scratchcountertwo
scratchdimen      : \meaningfull\scratchdimen
scratchdimenone   : \meaningfull\scratchdimenone
scratchdimentwo   : \meaningfull\scratchdimentwo
```

The meaning reveals what these are:

```
scratchcounter : permanent \count257
scratchcounterone : permanent \count260
scratchcountertwo : permanent \count261
scratchdimen : permanent \dimen257
scratchdimenone : permanent \dimen260
scratchdimentwo : permanent \dimen261
```

You can use the numbers directly but that is a bad idea because they can clash! In the original TEX engine there are only 256 registers and some are used by the engine and the core of a macro package itself, so that leaves a little amount for users. The ε-TEX extension lifted that limitation and bumped to 32K and LuaTEX upped that to 64K. One could go higher but what makes sense? These registers are taking part of the fixed memory slots because that makes nested (grouped) usage efficient and access fast. The number you see above is deduced from the so called command code (here indicated by `\count`) and an index encoded in the same token. So, `\scratchcounter` takes a single token contrary to the verbose `\count257` that takes four tokens where the number gets parsed every time it is needed. But those are details that a user can forget.

As mentioned, commands like `\newcount \foo` create a global control sequence `\foo` referencing a counter. You can locally redefine that control sequence unless in LuaMetaTEX you have so called overload mode enabled. You can do local or global assignments to these registers.

```
\scratchcounter = 123
\begingroup
    \scratchcounter = 456
    \begingroup
        \global\scratchcounter = 789
    \endgroup
\endgroup
```

And in both cases count register 257 is set. When an assignment is global, all current values to that register get the same value. Normally this is all quite transparent: you get what you ask for. However the drawback is that as a user you cannot know what variables are already defined, which means that this will fail (that is: it will issue a message):

```
\newcount\scratchcounter
```

as will the second line in:

```
\newcount\myscratchcounter
\newcount\myscratchcounter
```

In ConTEXt the scratch registers are visible but there are lots of internally used ones are protected from the user by more obscure names. So what if you want to use your own register names without ConTEXt barking to you about not being able to define it. This is why in LMTX (and maybe some day in MkIV) we now have local definitions:

```
\begingroup
  \newlocaldimen\mydimena     \mydimena1\onepoint
  \newlocaldimen\mydimenb     \mydimenb2\onepoint
  (\the\mydimena,\the\mydimenb)
  \begingroup
    \newlocaldimen\mydimena   \mydimena3\onepoint
    \newlocaldimen\mydimenb   \mydimenb4\onepoint
    \newlocaldimen\mydimenc   \mydimenc5\onepoint
    (\the\mydimena,\the\mydimenb,\the\mydimenc)
    \begingroup
      \newlocaldimen\mydimena \mydimena6\onepoint
      \newlocaldimen\mydimenb \mydimenb7\onepoint
      (\the\mydimena,\the\mydimenb)
    \endgroup
    \newlocaldimen\mydimend   \mydimend8\onepoint
    (\the\mydimena,\the\mydimenb,\the\mydimenc,\the\mydimend)
  \endgroup
  (\the\mydimena,\the\mydimenb)
\endgroup
```

The allocated registers get zero values but you can of course set them to any value that fits their nature:

```
(1.0pt,2.0pt)
(3.0pt,4.0pt,5.0pt)
(6.0pt,7.0pt)
(3.0pt,4.0pt,5.0pt,8.0pt)
(1.0pt,2.0pt)
```

You can also use the next variant where you also pass the initial value:

```
\begingroup
```

```
  \setnewlocaldimen\mydimena     1\onepoint
  \setnewlocaldimen\mydimenb     2\onepoint
  (\the\mydimena,\the\mydimenb)
  \begingroup
    \setnewlocaldimen\mydimena   3\onepoint
    \setnewlocaldimen\mydimenb   4\onepoint
    \setnewlocaldimen\mydimenc   5\onepoint
    (\the\mydimena,\the\mydimenb,\the\mydimenc)
    \begingroup
      \setnewlocaldimen\mydimena 6\onepoint
      \setnewlocaldimen\mydimenb 7\onepoint
      (\the\mydimena,\the\mydimenb)
    \endgroup
    \setnewlocaldimen\mydimend   8\onepoint
    (\the\mydimena,\the\mydimenb,\the\mydimenc,\the\mydimend)
  \endgroup
  (\the\mydimena,\the\mydimenb)
\endgroup
```

So, again we get:

(1.0pt,2.0pt)
(3.0pt,4.0pt,5.0pt)
(6.0pt,7.0pt)
(3.0pt,4.0pt,5.0pt,8.0pt)
(1.0pt,2.0pt)

When used in the body of the macro there is of course a little overhead involved in the repetitive allocation but normally that can be neglected.

4 Files

When adding these new allocators I also wondered about the read and write allocators. We don’t use them in ConTEXt but maybe users like them, so let’s give an example and see what more demands they have:

```
\integerdef\StartHere\numexpr\inputlineno+2\relax
\starthiding
SOME LINE 1
SOME LINE 2
SOME LINE 3
SOME LINE 4
\stophiding
\integerdef\StopHere\numexpr\inputlineno-2\relax

\begingroup
  \newlocalread\myreada
  \immediate\openin\myreada {lowlevel-scope.tex}
  \dostepwiserecurse{\StopHere}{\StartHere}{-1}{
    \readline\myreada line #1 to \scratchstring #1 : \scratchstring \par
  }
  \blank
  \dostepwiserecurse{\StartHere}{\StopHere}{1}{
    \read    \myreada line #1 to \scratchstring #1 : \scratchstring \par
  }
  \immediate\closein\myreada
\endgroup
```

Here, instead of hard coded line numbers we used the stored values. The optional `line` keyword is a LMTX speciality.

281 : SOME LINE 4
280 : SOME LINE 3
279 : SOME LINE 2
278 : SOME LINE 1

278 : SOME LINE 1
279 : SOME LINE 2
280 : SOME LINE 3
281 : SOME LINE 4

Actually an application can be found in a small (demonstration) module:

```
\usemodule[system-readers]
```

This provides the code for doing this:

```
\startmarkedlines[test]
SOME LINE 1
SOME LINE 2
SOME LINE 3
\stopmarkedlines
```

```
\begingroup
  \newlocalread\myreada
  \immediate\openin\myreada {\markedfilename{test}}
  \dostepwiserecurse{\lastmarkedline{test}}{\firstmarkedline{test}}{-1}{
    \readline\myreada line #1 to \scratchstring #1 : \scratchstring \par
  }
  \immediate\closein\myreada
\endgroup
```

As you see in these examples, we an locally define a read channel without getting a message about it already being defined.