LaTeX

Some notes on templates

Lars Hellström

Over the last few years, the LaTeX3 project team has been making and releasing some packages belonging to what they call LaTeX 2ε∗, which is a sort of intermediate step before LaTeX3. Unlike the previously released l3 suite of packages [2], which deals mainly with very basic programming structures such as lists and stacks, the LaTeX 2ε∗ suite of packages is more about producing better results in more concrete areas of LaTeX. Examples include the xor package, which is a new and much more versatile output routine, and the xparse package, with which one can easily define commands with complicated mixtures of mandatory, optional, and ∗-type arguments. The elegant interfaces and functionality of these packages promise well for the future.

Yet, most of them are still in a rather experimental state and they are currently only available from the experimental code directory on the LaTeX project web site [4]. Some of the packages are however not too far away from a more general release — in the event of which they will probably become part of the Required suite of LaTeX packages — and the first of these will most likely be the template package [1]. This is a very interesting package, because it provides the means for a whole new (and very promising) style of LaTeX programming, which I had the opportunity to try out during my work on the docindex package [3]. This note is an attempt to summarize the observations I've made about this programming style, in the hope that it may guide others in their first experiences of it. I am quite convinced it will become an important part of LaTeX3 and also of the further development of LaTeX 2ε.

Even though the name of it all is template, one shouldn't overrate the importance of the templates. Most of the things you actually keep around are not templates, but *instances*, and an instance is basically a very familiar object: a macro which performs some action. Of course, in TeX programming almost everything is a macro in one way or another, but some things are macros only because TeX doesn't provide any better way of storing some kinds of data, and some macros only exist to help other macros parse their arguments. Instances are rather the kind of macro you write because you want to modularize your code; typical actions are to typeset a float caption and to set the paragraph justification.

An instance is not just any nice collection of simpler commands, though. To begin with, every instance has a *type*, which specifies the syntax and most of the semantics of the instance. All instances with the same type do roughly the same thing, but they usually differ in the details. If one instance of a certain type, say, takes some text and typesets it as a section heading, then another instance of the same type could typeset the same text as a subsection heading or a part heading. If one instance of a certain type causes the index to be typeset (in the same sense that `\printindex` does) then another instance of that type might cause the glossary to be typeset. The simpler an instance is the more detailed the type specification usually gets, but it is generally about *what* the instance does, not about *how* it does it. Two instances of the same type are exchangeable in the sense that replacing one with the other doesn't cause any errors, although it will almost certainly change the typeset appearance of something.

That the instances are typed is incredibly useful, because it means you can redefine any instance without having to worry about breaking anything (as long as your redefinition conforms to the type specification)! An average LaTeX package usually comprises a couple of user level commands, a couple of parameters, and a number of private macros. The user level commands and parameters tend to have well-defined syntax and semantics, even though the choice of parameters is often less than satisfactory, but the syntax and semantics of the private macros are generally something about which everyone but the package author knows very little. There is often no other way to achieve a certain layout modification than to redefine a private macro, but that is always a risky operation because inferring the semantics of something from its implementation is no exact science.

In an implementation employing template techniques, many of the private macros would instead be instances, and thus this wouldn't be a problem; the type specification would say everything one needs to know. In the case of new types there is of course a bit of extra work needed for writing down the specification, but that work is usually well spent as it helps pointing out weaknesses in the implementation. An interesting side-effect of using instances is that there is much less need for package parameters, as many of these can instead be embedded into the instances which are easily redefined. Thus the net effect of using template techniques in a package can actually be that the interface to the package becomes simpler as well as more powerful and versatile.

The other thing about instances is how they are defined; this is where the templates get into the picture. A *template* is also basically a macro, and like an instance it has a type, but a template additionally has an associated set of parameters. Instances are defined by specifying a template (of the correct type) and the values that the parameters of the template should have. A typical instance definition looks something like

```
\DeclareInstance{justification}
  {flushleft}{std}{
    rightskip  =0pt plus 1fill,
    leftskip   =0pt,
    startskip  =0pt,
    parfillskip=0pt plus 1fill
}
```

Here `justification` is the type, `flushleft` is the name of the instance being defined, and `std` is the name of the template it is based on. The last argument is a keyval list of parameter names and values. What happens internally is that a couple of control sequences (the respective *storage bins* for the parameters of the template) are set to these values whenever the instance is used and the code in the template accesses the values by using the storage bins in very much the same way as we currently use a package parameter.

A parameter value need not be a length or some other numeric quantity, however; it can just as well be a function (which in this case is a fancy name for a command), a name, a boolean, or another instance (usually, but not necessarily, of a different type). A very common use for function valued parameters is to handle formatting of short pieces of text; for example, the heading of the `theorem` environment. A template which handles this might for example have a function valued parameter `heading-format` which receives the theorem number as its only argument. Then to get headings in the default "**Theorem 6.2**" style one could say

```
heading-format = \textbf{Theorem~#1}
```

whereas the reverse "**6.2 Theorem**" style would be the result of

```
heading-format =
    \textbf{#1\hspace*{0.5em}Theorem}
```

Generalizing slightly, one could also imagine there being a function parameter `named-heading-format` with two arguments which is used instead of the `heading-format` parameter when the theorem has a name (e.g., "Inverse Function Theorem"). Passing

this name as the second argument, a suitable value for `named-heading-format` in the first case might be

```
named-heading-format =
    \textbf{Theorem~#1 (#2)}
```

or even

```
named-heading-format =
    \textbf{Theorem~#1 (\textit{#2})}
```

whereas the second value for `heading-format` might go better with

```
named-heading-format =
    \textbf{#1\hspace*{0.5em}#2}
```

e.g., "**6.3 Inverse Function Theorem**".

I personally find the separation of code into on one hand a template and on the other hand values of the parameters of that template quite a relief, because it physically separates two different levels of programming. The actual template usually only contains the "hard", programming-like parts of the code — arithmetic, decision-making, interpretation of arguments, and so on — whereas "soft" parts like the layout specification are put in the parameters. This means whenever the code actually does something that is directly visible in the layout — such as insert a skip or format a heading — it simply uses the value in a parameter or passes the relevant data on to a parameter for further processing. The resulting code in the template looks very much like a skeleton of only the hard parts with some sprinkled markers saying "insert soft thing doing ... here", but it is actually complete and working. It cannot be used until the parameter values have been specified as well, though, and the normal way of doing this is to define an instance of the template.

Since the soft programming of selecting values for parameters is much more like writing a LaTeX document than the hard programming, it is not surprising that the many LaTeX users who have not mastered the hard programming will find that their ability to modify the behaviour of a package is much higher for templated packages than for traditional ones. For those who have mastered hard programming the advantages may seem less clear, but my personal experience was that programming became simpler. How can this be if I am actually restricting the ways in which I may write the code? I think it has to do with how one thinks about the problem at hand. When one is doing hard programming one also gets into a "hard" mode of thinking, whereas when one is doing soft programming one gets into a "soft" mode of thinking. In the traditional style the hard and soft parts are often heavily mixed and consequently one is forced to constantly switch between two modes of thinking. In the templated style the mixing is much less pronounced due to the aforementioned separation and consequently one does not have to switch mode that often. As it does require some effort for the mind to switch mode, the less one has to switch the better!

There are many other things which could be said about the template package — how one can use calc type expressions as parameter values, how one can use collections to effectively have several definitions of an instance in memory simultaneously and quickly switch between them, what one actually does to declare a new template — but these are things one can easily find in the template package manual. I certainly hope that you will give it a try, because this is one of these things after which LaTeX programming will never again be quite the same.

References

[1] David Carlisle and Frank Mittelbach. The template package. Available from `http://www.latex-project.org/code/experimental/template.tgz`, 1999.

[2] David Carlisle, Chris Rowley, and Frank Mittelbach. The LaTeX3 programming language — a proposed system for TeX macro programming. Available from CTAN, `macros/latex/exptl/project/expl3/`, 1998.

[3] Lars Hellström. The docindex package. Available from CTAN, `macros/latex/contrib/xdoc/docindex.dtx`, 2001.

[4] Various authors. LaTeX project web site directory for experimental code. Located at `http://www.latex-project.org/code/experimental/`, 1999–present.

◇ Lars Hellström
Sand 216
S-881 91 Sollefteå
SWEDEN
`Lars.Hellstrom@math.umu.se`