Extending the LaTeX templating mechanism

LaTeX Project*

v0.6f 2026-09-06

Abstract

This module contains code for the kernel implementation of templates (in `lttemplate.dtx`). After suitable testing this will be moved there.

Contents

*Initial implementation by Frank Mittelbach.

1 Handling ⟨*order*⟩ keys in a generic way

Templates often have to typeset several items of textual data (from keys and/or from template arguments) with slight variations from instance to instance, e.g., the ordering and the separation between the items might differ or in other cases only a subset of the items are present (or supported). For example, a theorem-like environment typically wants to have a fixed title, a number, some punctuation, and possibly a note (provided through document input). However, one design might ask for "Lemma 3.2 (note)", the next for "3.2 Lemma. (note)", and another for "3.2 Lemma *note*." with a different ordering and a punctuation somewhere or not.

While it is, of course, possible to achieve this by providing different templates that only differ in small aspects, it is often better to do this by providing a template key that defines the order of certain elements (and in which you can leave out some) and have the template code process this key and in this way achieve various layouts through a single template.

To make this possible such ⟨*order*⟩ keys and their allowed values need to follow a few conventions that we explain below and give some examples.

1.1 Conventions for the ⟨*order*⟩ comma list key

The ⟨*order*⟩ key has to be a clist and the corresponding variable name (i.e., the binding) must be `\l_@@_`⟨*order*⟩`_clist`, e.g., in `\DeclareTemplateInterface`

```
caption-order : commalist = { title, separator-a,
                              number, separator-b, note, punct }
```

and in `\DeclareTemplateCode`

```
caption-order = name {l_@@_caption-order_clist}
```

The clist variable needs to contain the ⟨*order*⟩ key name which is why we had to resort to the `name` interface to get a nonstandard variable name containing a `-`. For simple key names like `order`, that would not be necessary and the line would look like this:

```
order = \l_@@_order_clist
```

Of course, there is no requirement to provide a default value (as we did above) but in many cases there is a commonly used sequence that could be provided as a default.

1.2 Conventions for the items in the ⟨*order*⟩ key clist

For each ⟨*item*⟩ in the comma list, e.g, in the above example `title`, `separator-a`, `number`, `punct`, `separator-b`, and `note`, there has to exist a variable with the name `\l_@@_`⟨*item*⟩`_tl`. If the items themselves are keys in the template that is achieved through settings such as

```
title       : tokenlist = Lemma ,
separator-a : tokenlist = \enspace ,
...
```

and

```
title       = \l_@@_title_tl ,
separator-a = name {l_@@_separator-a_tl} ,
...
```

Again, we had to make use of the `name` keyword because the key `separator-a` contains a hyphen.

If the data of such an item is user input and provided through a mandatory argument to the template, e.g., the data for a `note`, then you can fulfill the requirement by defining the variable in the template code, e.g.,

```
\tl_set:Nn \l_@@_note_tl {#3}
```

or in whatever argument the note data is passed to the template.

If the user hasn't provided a note, then this is indicated by passing `\NoValue` to the template in that argument (as long as `\NewDocumentCommand` or a similar extended command definition offered by the LaTeX kernel is used). In that case the mechanism correctly interprets this and ignores a `note` item in the processing of the ⟨*order*⟩ key.

The existence of `\l_@@_`⟨*item*⟩`_tl` is required: if it is not defined, then using ⟨*item*⟩ in the comma list will lead to an error message.

Not required, but often wanted, are the following additional token list variables and commands (per ⟨*item*⟩):

`\l_@@_`⟨*item*⟩`_decls_tl` A token list that contains declarations (such as font changes) that are applied just before the ⟨*item*⟩ data is typeset. Processing happens in a group, so that changes are reverted after the ⟨*item*⟩ was typeset.

`\@@_`⟨*item*⟩`_format:n` A command that receives the ⟨*item*⟩ data as its argument and can then manipulate it prior to typesetting.

So in summary what gets typeset is

```
\group_begin:
  \l_@@_⟨item⟩_decls_tl
  \@@_⟨item⟩_format:n { \l_@@_⟨item⟩_tl }
\group_end:
```

Note that this means that paragraph parameters (such as `\baselineskip`) that are only evaluated at the end of a paragraph are ignored if placed in `\l_@@_`⟨*item*⟩`_decls_tl` or `\@@_`⟨*item*⟩`_format:n` unless the latter contains a `\par` that ends the current paragraph.[1]

Strictly speaking, the `\l_@@_`⟨*item*⟩`_decls_tl` token list is not needed, since all one can do with it can also be done through a suitably defined `\@@_`⟨*item*⟩`_format:n` command. Thus, what to use is often a matter of taste. Our default templates provide both to cater for different preferences.

If the token list variable and command are exposed via keys (so that they can be set in an instance) then the names have to be given exactly as specified above in order for the mechanism to pick them up. However, you are free to choose whatever key name you like. By convention, we use ⟨*item*⟩`-decls` and ⟨*item*⟩`-format` as key names because we always use `-` in longer key names.

[1] This is different from separator items that are executed outside of these groups. This means that parameter changes done in a separator are seen by all following ⟨*item*⟩s.

1.3 Conventions for the separators used in the ⟨*order*⟩ key clist

Besides items containing textual data there is also often the need to specify data that separates them (typically by some space). These separators need a somewhat different handling, because if an item such as a `note` is not present, then a separator before it should normally be dropped, e.g., one does not want to get "Lemma 3.2␣." but "Lemma 3.2." if the punctuation key `punct` comes at the end of the ⟨*order*⟩ list and the `note` before it was dropped.

Similarly, in some situations one wants to drop a separator after a data item that has no value, that can be done too, by combining the separator with the special `?` or `|` items discussed below.

To support this behavior, it is necessary for the mechanism to identify that a ⟨*item*⟩ name is a separator and not a normal textual item. We therefore recommend to use (some of) the following names that have already been prepared for use with the mechanism.[2]

So a typical setup is

```
separator-a : tokenlist = \ ,
separator-b : tokenlist = \ ,
...

separator-a = name {l_@@_separator-a_tl} ,
separator-b = name {l_@@_separator-b_tl} ,
...
```

up to `separator-e` and for cases where you need only one separator there is also the key `separator` (which can, of course, also be used in addition to the others).

If several separators are specified directly after another in the ⟨*order*⟩ clist then all of them are typeset or dropped depending on the next item that isn't a separator. Separator(s) at the very start of the clist use the same logic.

If a separator is specified at the very end of the clist it is always typeset (unless the special item | is used after it in which case it is only typeset if the preceding data item had a value).

If you need some separation that should not be dropped in case the next normal item is absent (i.e., has `\NoValue` as its value) then you can use `|` after it to control it from the previous data item or use `!` to typeset it unconditionally.

1.4 The special items `?`, `|`, and `!` in the ⟨*order*⟩ key clist

Separators *before* an item that has `\NoValue` as its value are dropped, but in some cases some or all of the separators *following* such an item should also be dropped. For example, in the order list

```
order = {prefix, separator-a, number, ...
```

one normally doesn't want a space in front of the `number` if there is no `prefix`. To implement such scenarios easily, one can use the special item `?`, which has the following effect: it drops all collected separators if the last data item was dropped, otherwise it does nothing and the decision what to do with the collected separators happens when the next data item is processed. For example, if we have

[2] It is possible to use other names or provide more with the help of `\template_new_order_separator:n` if really necessary, but the five (or rather six) we offer should normally be sufficient.

```
order = {prefix, separator-a, ? , separator-b, number, ...
```

and `prefix` was `\NoValue` then `separator-a` is dropped but `separator-b` after the `?` is used unless `number` is also `\NoValue`. If `prefix` has a value but `number` has no value, then both separators are dropped.

The `|` special item is similar but in that case separators on the left of it are fully controlled by the previous data item, i.e., if that item has no value they are dropped and if it has a value then they are directly typeset and thus independent of the next data item.

Finally, there is also a `!` special item which results in all collected separators to be unconditionally typeset, i.e., `separator,!` behaves as if the separator is a normal data item (except that separators do not have corresponding `...-decls` or `...-format` keys for customization).

For even more granular control, there is also `\IfValueLastOrderItemTF` which can be used inside a separator (or even inside the value of a normal data item) to produce different actions depending on whether or not the last data item had a value.

1.5 Grouping items in the ⟨*order*⟩ key clist

In some cases it is necessary to group some items in the ⟨*order*⟩ clist, e.g., to indicate that the `prefix` and the `number` form the caption label when it comes to tagging structures. This is done by using special items in the clist: `<`⟨*name*⟩ to start a group and ⟨*name*⟩`>` to end it. For example

```
order = { <label, prefix, separator-a, number, label>, ... }
```

would group `prefix`, `separator-a`, and `number`.

These special group items have to be declared using the declaration `\template_new_order_group:nnnnnn`. With such a declaration they are made known to the order key processing mechanism. In addition, the necessary tagging support code is defined, e.g., in the above example tagging support would probably add a `<Lbl>` structure.

Even though we may want to use `label` as a group ⟨*name*⟩ in different kinds of templates, the underlying tagging support code will most likely differ from case to case. The declaration therefore takes the current ⟨*module*⟩ as its first argument and ⟨*name*⟩ as its second argument.[3] The remaining four arguments define what should happen when `<`⟨*name*⟩ and ⟨*name*⟩`>` are processed. Details are given in the implementation section.

To specify specific (typesetting) declarations for such a group the token list `\l_@@_`⟨*name*⟩`_decls_tl` is available. Thus, the template can set up, for example

```
label-decls : tokenlist = ,
```

and

```
label-decls = \l_@@_label_decls_tl ,
```

to enable customization during instance declaration. If no customization is desired then just don't expose the variable in which case it automatically remains empty. Note that `\@@_`⟨*name*⟩`_format:n` *is not supported* for such groups.

Examples for such group items can already be found in `thmstyle` templates and in `caption` templates. More will follow over time.

[3] This may need extension or change, e.g., perhaps we need to make it based on the current template type and template name to achieve a proper separation, but for now we hope that ⟨*module*⟩ is enough. This is quite different to the case of separators which can be reused across all templates because they only need to pass their values to the order key processing mechanism, i.e., their definition is always the same.

1.6 Processing the ⟨*order*⟩ key in the template code

In the template code you use the ⟨*order*⟩ key in the following way

```
\template_process_order_clist:nnn
   { ⟨module⟩ }
   { ⟨order key name⟩ }
   { ⟨supported items⟩ }
```

where first argument is the current module name (i.e., what `@@` produces but without the leading `__`) and the second argument is the name of your order key to process. The third argument is a comma list of support items, i.e., the value of the ⟨*order key name*⟩ must contain only items from that argument.[4]

This command then typesets the items listed in the ⟨*order*⟩ key or more precisely the content stored in the associated token list parameters `\l_@@_`⟨*item*⟩`_tl`, applying `\l_@@_`⟨*item*⟩`_decls_tl` and/or `\@@_`⟨*item*⟩`_format:n` if defined, and ignoring those items that have `\NoValue` as their token list value. Separators in front of ignored items are dropped, all others are typeset in the specified places unless the list contains any special items `?`, `|`, or `!` in which case they control what happens with the separators.

The command also handles the tagging, i.e., it adds the necessary structures. How this can be influenced is described below.

1.7 Tagging produced when processing the ⟨*order*⟩ key

If nothing special is specified then the order key processing mechanism typesets the items without adding any specific tagging structures. It does, however, ensure that everything is wrapped into one or more MCs (marked content structures). This is normally automatically done by LaTeX's paragraph tagging. However, in situations where the mechanism is applied, we often need `\tagpdfparaOff` in which case switching in and out of hmode does not generate MCs on its own. The `\template_process_order_clist:nnn` command therefore

- checks and remembers if an MC is already open at the start;
- if not, it opens one;
- it then processes all items, possibly also adding structure elements as explained below;
- and at the end it restores the MC state it had found at the beginning, e.g., closes the MC if none was open at the start. However, if it ends in vertical mode it does not reopen an MC if one was open when it started.

In addition it is also possible to surround individual items from the ⟨*order*⟩ list with a structure element, e.g., a `<text-fragment>` or an `<Artifact>`. In this case, the current MC is ended, the structure and a new MC are started, then the item is typeset. Afterwards the MC and the structure are closed and the next item in the list then opens a new MC to be used for the remainder of the ⟨*order*⟩ list.[5]

[4] This is checked to prevent the use of items that aren't supported by the current template but are defined by some other template and would therefore produce unpredictable and erroneous results.

[5] Perhaps a better approach would be to use dedicated sockets generated from the ⟨*item*⟩ name. That would give more flexibility that may be needed.

For this to work the desired structure name needs to be stored in the token list `\l_@@_⟨item⟩_tag_tl`. This can be hardwired in the template code or offered through a key, if the need for customization is expected, e.g.

```
prefix-tag-name : tokenlist ,
```

and

```
prefix-tag-name = \l_@@_prefix_tag_tl ,
```

and then changed with `prefix-tag-name = Artifact` in an instance.

As already mentioned, the grouping items also produce special tagging. For example, an order key setting for a caption might look like

```
order = { <label , ... , label> , <data , ... , data> }
```

where the `label` group generates a `<Lbl>` structure element (with paragraph tagging off and MC handling inside managed by the order key processing), and the `data` group that generates one or more `<text-block>` structures (aka. `<P>`) and may also contain other structures such as lists or quotations.

To make this possible, the `\template_new_order_group:nnnnnn` declaration defines a normal and a tagging socket for use in `<`⟨*name*⟩ and another pair for use in ⟨*name*⟩`>`. The names are ⟨*module*⟩`/`⟨*name*⟩`/begin` and `.../end`, respectively. The normal sockets are usually empty, but may contain a `\leavevmode` or a `\par` (or something else) if tagging is handled through normal paragraph processing and a defined state at the start or end is needed. Most of the time, the tagging payload is only in the tagging sockets.

2 Interfaces

`\template_process_order_clist:nnn` `\template_process_order_clist:nnn {⟨module⟩} {⟨order key name⟩} {⟨supported items⟩}`

Takes an ⟨*order*⟩ key and processes its items, i.e., typesets them and adds tagging structures.

The current ⟨*module*⟩ is explicitly given, i.e., what `@@` would produce but without the two underscores as prefix, e.g., `thmstyle`. It is used to construct related variables and is needed in error messages. The ⟨*order key name*⟩ is the name of the key you want to process. Finally, the ⟨*supported items*⟩ argument lists the items that are allowed to appear in the value of the order key (except for `?`, `|`, and `!` that do not need to be listed explicitly). Thus a typical call would be

```
\template_process_order_clist:nnn
  {thmstyle}
  {order}
  {note,number,punct,separator,title}
```

\template_new_order_separator:n `\template_new_order_separator:n{`⟨*separator item name*⟩`}`

Separators are items in an order clist that are supposed to vanish if the next normal item is not present (typically they generate some space). Their names need to be predeclared to have that role which is done with `\template_new_order_separator:n`. Once declared the ⟨*separator item name*⟩ has this role in any order key, i.e., in contrast to group items the declaration applies globally and not only to names in a specific module.

A suitable number of names are predeclared, so that function should be seldom needed.

\template_new_order_group:nnnnnn `\template_new_order_group:nnnnnn {`⟨*module*⟩`} {`⟨*group*⟩`}`
`{`⟨*begin normal socket code*⟩`} {`⟨*begin tagging socket code*⟩`}`
`{`⟨*end normal socket code*⟩`} {`⟨*end tagging socket code*⟩`}`

This declares a new pair of ⟨*group*⟩ items in ⟨*module*⟩ and declares sockets with the names ⟨*module*⟩/⟨*group*⟩/`begin` and ⟨*module*⟩/⟨*group*⟩/`end`, both as normal and as tagging sockets. It also defines plugs with the name `default` for each of them from the content of the arguments 3–6.

Further plugs can be defined through `\NewSocketPlug` or `\NewTaggingSocketPlug` declarations and plugged in when necessary.

3 Other code updates/extensions

`\SetKnownTemplateKeys` is updated to do nothing (i.e., not start key parsing)) if the argument is empty or contains `\NoValue` (after one expansion). These are typical scenarios, e.g., when a key list is passed in as `\UnusedTemplateKeys`. If the argument contains a key/value list without any indirection then an `o`-expansion will expand the first letter of the first key so no harm is done.

`\SetTemplateKeys` was changed to use `\SetKnownTemplateKeys` and then test `\UnusedTemplateKeys` instead of using the low-level mechanism, because the latter exposes internals of the implementation in its error message when an unknown key is encountered as it shows the full path of the key.

The command `\__template_debug_typeout:n` was made public (as `\template_debug_typeout:n` because any template may want to issue debugging messages that are activated when `\DebugTemplatesOn` is used.

The conditional `\IfTemplateExistsTF` and its variants have been added so that classes or packages can check if a certain template is provided and then declare instances based on it and if not do something else.

4 The Implementation

This code will eventually move to `lttemplate.dtx`, for now it is only available when `\DocumentMetadata` is used.

```
⟨*package⟩
⟨@@=template⟩

\ProvidesPackage {latex-lab-testphase-template}
                 [\ltlabtemplatedate\space v\ltlabtemplateversion\space
                           template implementation]
\ExplSyntaxOn
```

`\checkstatus` A temporary debugging helper.

```
\cs_new_protected:Npn \checkstatus #1 {
  \__template_debug_typeout:n{-------------------- #1 ----------------}
  \mode_if_horizontal:TF
       { \__template_debug_typeout:n{==>~hmode} }
       { \__template_debug_typeout:n{==>~vmode} }

  \__template_debug_typeout:n{==>~para-tagging:~
         \bool_if:NTF \l__tag_para_bool
     { on~ (using~ \l__tag_para_tag_tl)}
     { off }
                  }
  \tag_mc_if_in:TF
      { \__template_debug_typeout:n{\@spaces mc~ status:~ open} }
      { \__template_debug_typeout:n{\@spaces mc~ status:~ closed} }
  \__template_debug:n { \ShowTagging{struct-stack=log } }
%  \__template_debug:n { \ShowTagging{struct-stack } }
}
```

(*End of definition for* `\checkstatus`*. This function is documented on page* **??**.)

`\template_process_order_clist:nnn` The processing interface inside template code.

```
\cs_new_protected:Npn \template_process_order_clist:nnn #1#2#3 {
  \__template_debug_typeout:n { Processing~ order~ key~ '#2'}
  \checkstatus{Start}

  \group_begin:
```

Setting up the command to restore the MC state after processing. This can be shortened a lot if the debugging code gets removed and then it could probably use the push/pop commands.

drop the debugging eventually

```
  \tag_mc_if_in:TF
      { \cs_set_protected:Npn \__template_restore_mc_state: {
          \mode_if_horizontal:TF
            {
              \tag_mc_if_in:TF
                  { \__template_debug_typeout:n {\@spaces mc~ restore~
                      already~open}
                  }
                  { \__template_debug_typeout:n {\@spaces mc~ restore~ to~ open}
                    \tag_mc_begin:n{}
                  }
            }
```

Verify logic, I haven't fully convinced myself!

Do not restore an initially open MC if we end up in vertical mode after the order key processing. That means there was some `\par` along the way and we assume that this did the closing (which would be the correct state then)

```
            { \__template_debug_typeout:n {\@spaces no~ mc~ restore~ in~
                vertical~ mode }
            }
        }
      }
      { \cs_set_protected:Npn \__template_restore_mc_state: {
          \tag_mc_if_in:TF
              { \__template_debug_typeout:n {\@spaces mc~ restore~ to~ closed}
                \tag_mc_end:
              }
              { \__template_debug_typeout:n {\@spaces mc~ restore~
                                                    already~closed}
              }
        }
      }
```

To support the special `?`, `|`, and `!` values in any order list we need to announce it to the algorithm. The flag used for this is a token list that has the current module in its name, so we do this here, because there is no point in making the template writer add this.

```
\tl_clear_new:c { l _ _ #1_?_tl }
\tl_clear_new:c { l _ _ #1_|_tl }
\tl_clear_new:c { l _ _ #1_!_tl }
```

Also initialize the boolean that tells us if the last data item had a value (start of the list is considered to be an invisible item with a value).

```
\bool_gset_true:N \g__template_order_item_value_bool
```

Loop through the comma list stored in the order key and process it item by item. The order key variable is `\l__#1_#2_clist` but we write it as `l␣␣␣#...` in the code to avoid bogus warnings from `l3doc` about using private names from other modules. We do the same for other construct variables. Argument `#3` holds the list of allowed items to which we add `?`, `|`, and `!` because they should be always allowed.

```
\clist_map_inline:cn {l _ _ #1_#2_clist }
      { \clist_if_in:nnTF { ?, | , ! , #3 } { ##1 }
         {
```

For each clist item the token list `\l__`⟨*module*⟩`_`⟨*item*⟩`_tl` must exist! If it does then the clist item is processed with `\__template_process_order_key:NNnnn` which receives 2 token lists (pre-constructed for efficiency) as well as the module name, the clist item name, and the order key name from which it came from.

```
            \cs_if_exist:cTF { l _ _ #1_##1_tl }
               { \exp_args:Ncc
                 \__template_process_order_key:NNnnn
                   { l _ _ #1_##1_tl }
                   { l _ _ #1_##1_tag_tl }
                   { #1 }
                   { ##1 }
                   { #2 }
               }
```

If the token list doesn't exist we raise an error:

```
                { \msg_error:nnnn { template } { unknown-key-value }
                                  { #2 } { ##1 }
                }
            }
```

If the item was not in the allowed list (i.e., `?,|,!,#3`) we raise the same error.

```
            { \msg_error:nnnn { template } { unknown-key-value }
                              { #2 } { ##1 }
            }
        }
```

In case the clist ended in separator items we insert those now as that hasn't happened yet.

```
  \__template_insert_collected_separators:
```

Then we restore the MC state that was current at the beginning.

```
  \__template_restore_mc_state:
  \group_end:
  \checkstatus{End}
```

Finally we clear all token lists that may have been set up by the template and used the processing. This avoids that we get bogus matches in other templates that also use the item name but without setting up default declarations for customization.

```
  \clist_map_inline:nn {#3}
    {
```

The list may contain group items starting with `<` and for them we have to get rid of the first character. For group items ending in `>` nothing needs to be done.

```
      \tl_if_head_eq_meaning:nNTF {##1} <
        {
          \tl_clear_new:c   { l _ _ #1_\use_none:n ##1_decls_tl }
        }
        {
          \tl_if_in:nnF {##1} >
            {
```

We actually clear a bit too much as this also loops through all the separators but it is simply to clear a few additional token lists rather than to figure out if the current `##1` is a name set up as a separator.

maybe change that.

```
              \tl_clear_new:c { l _ _ #1_##1_decls_tl }
              \tl_clear_new:c { l _ _ #1_##1_tag_tl }
              \cs_set_eq:cN   { l _ _ #1_##1_format:n } \use:n
            }
        }
    }
}
\msg_new:nnnn { template } { unknown-key-value }
  { The~ value~ '#2'~ in~ key~ '#1'~ is~ not~ recognized. }
  {
    Perhaps~ a~ misspelling~ or~ the~ current~ template~
    instance~ uses~ special~ values.
  }
```

(*End of definition for* `\template_process_order_clist:nnn`. *This function is documented on page 7.*)

`\__template_restore_mc_state:` Internal command to keep track of the MC state before and after order key processing.

```
\cs_new_protected:Npn \__template_restore_mc_state:{}
```

(*End of definition for* `\__template_restore_mc_state:`.)

`\g__template_order_item_value_bool` We keep track of whether or not the last data item had a value through a global boolean.

```
\bool_new:N \g__template_order_item_value_bool
```

(*End of definition for* `\g__template_order_item_value_bool`.)

`\IfValueLastOrderItemTF` `\IfValueLastOrderItemT` `\IfValueLastOrderItemF` Conditionals for use in separator values (or even data item values) to do some conditional processing based whether or not the last item had a value.

```
\cs_new_protected:Npn \IfValueLastOrderItemTF {
  \bool_if:NTF \g__template_order_item_value_bool
}
\cs_new_protected:Npn \IfValueLastOrderItemT {
  \bool_if:NT \g__template_order_item_value_bool
}
\cs_new_protected:Npn \IfValueLastOrderItemF {
  \bool_if:NF \g__template_order_item_value_bool
}
```

(*End of definition for* `\IfValueLastOrderItemTF`, `\IfValueLastOrderItemT`, *and* `\IfValueLastOrderItemF`. *These functions are documented on page* **??**.)

`\template_new_order_separator:n` Declare a new separator item that can be used with any order key. They are all mapped to run `\__template_process_special_order_separator:nn`.

```
\cs_new_protected:Npn \template_new_order_separator:n #1 {
  \cs_new_eq:cN { __template_process_special_order_#1:nn }
                \__template_process_special_order_separator:nn
}
```

(*End of definition for* `\template_new_order_separator:n`. *This function is documented on page 7.*)

`template_process_special_order_separator:nn` Processing a separator item simply means that we append its content to `\g__template_collected_separators_tl`.
This implicitly defines the separator called `separator`.

```
\cs_new:Npn \__template_process_special_order_separator:nn #1#2 {
    \__template_debug_typeout:n {\@spaces collecting~ separator~ data }
    \tl_gput_right:Ne \g__template_collected_separators_tl
       { \exp_not:c { l _ _ #1_#2_tl } }
}
```

(*End of definition for* `\__template_process_special_order_separator:nn`.)

We predefine a number of additional separators.

```
\template_new_order_separator:n {separator-a}
\template_new_order_separator:n {separator-b}
\template_new_order_separator:n {separator-c}
\template_new_order_separator:n {separator-d}
\template_new_order_separator:n {separator-e}
```

`\g__template_collected_separators_tl` The global tokenlist in which we collect separator data during the processing.

```
\tl_new:N \g__template_collected_separators_tl
```

(*End of definition for* `\g__template_collected_separators_tl`.)

`\__template_insert_collected_separators:` Inserting collected separators is done by typesetting the content of `\g__template_collected_separators_tl` if it is non-empty.

```
\cs_new:Npn \__template_insert_collected_separators: {
  \tl_if_empty:NF
      \g__template_collected_separators_tl
      {
        \__template_debug_typeout:n {\@spaces
          using~ collected~ separator~ data }
```

If we aren't already inside an MC we may have to start one and then typeset what was collected:

```
        \__template_open_mc_if_necessary:n { \@spaces opening~ mc~ for~
                                             separator }
        \g__template_collected_separators_tl
```

If in horizontal mode we also add a fake pdf space to ensure that the separator is seen as a space of some sort in the tagging structure.

```
        \mode_if_horizontal:T \pdffakespace
```

Finally we clear the tokenlist.

```
        \tl_gclear:N \g__template_collected_separators_tl
      }
}
```

(*End of definition for* `\__template_insert_collected_separators:`.)

`\__template_drop_collected_separators:` Dropping collected separators simply means clearing the `\g__template_collected_separators_tl` token list.

```
\cs_new:Npn \__template_drop_collected_separators: {
  \tl_if_empty:NF \g__template_collected_separators_tl
     {
       \__template_debug_typeout:n {\@spaces
                                dropping~ collected~ separators }
       \tl_gclear:N \g__template_collected_separators_tl
     }
}
```

(*End of definition for* `\__template_drop_collected_separators:`.)

`\__template_open_mc_if_necessary:n` Open an MC if none is open and it isn't automatically done when switching to hmode. The argument is a debugging/info message.

```
\cs_new:Npn \__template_open_mc_if_necessary:n #1 {
```

If `\l__tag_para_bool` is false we have to always open an MC if none is open. However, if this is true, i.e., the MC is handled by the paragraph builder we only have to do it if we are already in horizontal mode but no MC is open for some reason.

```
  \tag_mc_if_in:TF
    { \__template_debug_typeout:n {\@spaces mc~ already~ open} }
    { \bool_if:NTF \l__tag_para_bool
        { \mode_if_horizontal:T
            {
              \tag_mc_begin:n {}
```

```
              \__template_debug_typeout:n { #1 }
            }
        }
        { \tag_mc_begin:n {}
          \__template_debug_typeout:n { #1 }
        }
    }
}
```

(*End of definition for* `\__template_open_mc_if_necessary:n`.)

`\__template_process_order_key:NNnnn` The heart of the processing: dealing with one item. Arguments are

- Token list variable holding the item data (or `\NoValue`)
- Token list variable holding the item tag name (or empty or `\relax`)
- Current module name
- Current item to be processed
- Name of the order key being processed

```
\cs_new_protected:Npn \__template_process_order_key:NNnnn #1#2#3#4#5 {
  \__template_debug_typeout:n { Processing~ key~ '#4'~ in~ '#5' }
```

There are two types of items, either normal ones or special ones (separators or group items). The latter can be identified by checking if the command `\__template_process_special_order_` ⟨*item*⟩`:nn` exists.[6]

```
  \cs_if_exist:cTF { __template_process_special_order_ #4 :nn }
```

If we have a special item we call its processing function and pass it the current module and the current item name, i.e., `#3` and `#4`.

```
    { \__template_debug_typeout:n{\@spaces special~group~or~separator~item}
      \use:c  { __template_process_special_order_ #4 :nn } {#3}{#4}
    }
```

If we have a normal item then its value may be `\NoValue` in which case we ignore it and also drop any collected separators.

```
    { \tl_if_novalue:oTF #1
        { \__template_debug_typeout:n {\@spaces \exp_not:N \NoValue  }
          \__template_drop_collected_separators:
```

Current item has no value so update the boolean tracking that.

```
          \bool_gset_false:N \g__template_order_item_value_bool
        }
```

If the item has a real value (including an empty one) we insert all collected separators and then typeset the item in a group to keep formatting changes local.

```
        { \__template_insert_collected_separators:
          \group_begin:
```

[6] This means that you can't have an item being a normal one in one template but a special one in another, but — tough.

The token list variable for the tag name in #2 may not exist, in which case it was set to \scan_stop: (aka. \relax) by TEX in the \cs:w operation generating the argument. We therefore make a quick test and declare it as a token list variable, if necessary.

```
\cs_if_eq:NNT #2 \scan_stop:
  { \tl_new:N #2 }
```

We can now be sure that the tag name variable is either empty (no tag) or contains a tag name. If it is empty all we have to do is to ensure that we have an open MC when typesetting starts.

```
\tl_if_empty:NTF #2
  { \__template_open_mc_if_necessary:n { \@spaces opening~ mc~
                                         for~ '#4'} }
```

Otherwise, we have to build a structure around the item data. Thus, we first have to close an open MC (if one is open).

```
  { \tag_mc_if_in:T
    { \__template_debug_typeout:n { \@spaces closing~ mc~ before~
                                    struct~ '#2' }
      \tag_mc_end:
    }
    \tag_struct_begin:n { tag=#2 }
    \tag_mc_begin:n {}
    \__template_debug_typeout:n { \@spaces opening~ mc~ inside~
                                  struct~ '#2'  }
  }
```

The token list with declarations may not exist and the same might be true for the formatting command for the current item. If that's the case we declare them ...

```
\tl_if_exist:cF { l _ _ #3_#4_decls_tl }
   { \__template_debug_typeout:n { \@spaces #4-decls~ not~ defined~
                                   by~ the~ template }
     \tl_new:c { l _ _ #3_#4_decls_tl }
   }
\cs_if_exist:cF { _ _ #3_#4_format:n }
   { \__template_debug_typeout:n { \@spaces #4-format~ not~
                                   defined~ by~ the~ template }
     \cs_new_eq:cN  { _ _ #3_#4_format:n } \use:n
   }
```

... and then typeset the item data stored in #1:

```
\use:c { l _ _ #3_#4_decls_tl }
\__template_debug_typeout:n {\@spaces typeset~ '#4'~ data }
\use:c {  _ _ #3_#4_format:n } { #1 }
```

Finally, we have to close the structure if we opened one earlier.

```
\tl_if_empty:NF #2
    { \tag_mc_end: \tag_struct_end:
      \__template_debug_typeout:n { \@spaces reopening~ mc~ after~
                                    struct~ '#2' }
      \tag_mc_begin:n {}
    }
\group_end:
```

Current item had a value so update the boolean tracking that.

```
            \bool_gset_true:N \g__template_order_item_value_bool
\checkstatus{After~ #4}
          }
     }
}
```

(*End of definition for* `\__template_process_order_key:NNnnn`.)

`\__template_process_special_order_?:nn` We need this internal command to tell the algorithm that `?` is a special item in the order list.
It checks if collected separators should be dropped and if so does that.

```
\cs_set:cpn { __template_process_special_order_?:nn } #1#2 {
  \bool_if:NF \g__template_order_item_value_bool
    { \__template_drop_collected_separators: }
}
```

(*End of definition for* `\__template_process_special_order_?:nn`.)

`\__template_process_special_order_|:nn` The special item `|` similar but immediately typesets the collected separators if the previous data item had a value.

```
\cs_set:cpn { __template_process_special_order_|:nn } #1#2 {
  \bool_if:NTF \g__template_order_item_value_bool
    { \__template_insert_collected_separators: }
    { \__template_drop_collected_separators:   }
}
```

(*End of definition for* `\__template_process_special_order_|:nn`.)

`\__template_process_special_order_!:nn` Thee special item `!` unconditionally typesets the separators.

```
\cs_set:cpn { __template_process_special_order_!:nn } #1#2 {
  \__template_insert_collected_separators:
}
```

(*End of definition for* `\__template_process_special_order_!:nn`.)

`\template_new_order_group:nnnnnn` Declare a pair of group items <⟨*#2*⟩ and ⟨*#2*⟩>.
For this, we first define `\__template_process_special_order_<#2:nn`:

```
\cs_new_protected:Npn \template_new_order_group:nnnnnn #1#2#3#4#5#6 {
  \cs_set:cpn { __template_process_special_order_<#2:nn } ##1##2 {
```

Start by inserting any collected separators up to this point.

```
    \__template_insert_collected_separators:
```

Then start a group and process the declaration for the group.

```
    \group_begin:
      \tl_use:c { l _ _ #1_#2_decls_tl}
```

Then run the normal and the tagging sockets for `#1/#2/begin`.

```
      \UseSocket{X-#1/#2/begin}
      \__template_debug_typeout:n {\@spaces use~tagging~socket~#1/#2/begin }
      \UseTaggingSocket{X-#1/#2/begin}                % X- for now because of
                                                      % existing
                                                      % sockets from
                                                      % old implementation -
                                                      % needs cleanup
  }
```

When we reach the end group item we run this command:

```
\cs_set:cpn { __template_process_special_order_#2>:nn } ##1##2 {
```

Again, first typeset any collected separators, then the sockets and finally closing the group:

```
      \__template_insert_collected_separators:
      \UseSocket{X-#1/#2/end}
      \__template_debug_typeout:n {\@spaces use~tagging~socket~#1/#2/end }
      \UseTaggingSocket{X-#1/#2/end}
    \group_end:
    \checkstatus{#2~end}
}
```

Here are the socket declarations:

```
\NewSocket{X-#1/#2/begin}{0}
\NewSocket{X-#1/#2/end}{0}
\NewTaggingSocket{X-#1/#2/begin}{0}
\NewTaggingSocket{X-#1/#2/end}{0}
```

And these are the default plug settings:

```
\NewSocketPlug{X-#1/#2/begin}{default}{#3}
\NewTaggingSocketPlug{X-#1/#2/begin}{default}{#4}
\NewSocketPlug{X-#1/#2/end}{default}{#5}
\NewTaggingSocketPlug{X-#1/#2/end}{default}{#6}

\AssignSocketPlug{X-#1/#2/begin}{default}
\AssignTaggingSocketPlug{X-#1/#2/begin}{default}
\AssignSocketPlug{X-#1/#2/end}{default}
\AssignTaggingSocketPlug{X-#1/#2/end}{default}
```

And here is the token list for the declarations.

```
\tl_new:c { l _ _ #1_#2_decls_tl }
```

We also need the following two token lists to be defined to keep the algorithm happy: they serve as a flag that these items are allowed in the order key value.

```
\tl_new:c { l _ _ #1_<#2_tl }
\tl_new:c { l _ _ #1_#2>_tl }
}
```

(*End of definition for* `\template_new_order_group:nnnnnn`*. This function is documented on page 8.*)

5 Other code updates/extensions

5.1 Augmented `\SetKnownTemplateKeys`

`\SetKnownTemplateKeys`

A key/val list passed to `\SetKnownTemplateKeys` can either be empty (in which we do not want to start up the parsing machinery) or it could be `\NoValue` in which we do not want to do that either. The latter can happen, for example, with `verbatim` where we define the optional argument with `={late-legacy-code} !o` so that people can write `\begin{verbatim}[\small]` a syntax promoted by the TUGboat class.

```
\cs_set_protected:Npn \SetKnownTemplateKeys #1#2#3
  {
```

An "empty" argument (or rather one that is empty after one expansion) is the most likely case so we test for this first.

```
    \tl_if_empty:oTF {#3}
      {
        \tl_set_eq:NN  \UnusedTemplateKeys \c_empty_tl
      }
      {
        \tl_if_novalue:nTF {#3}
          {
            \tl_set_eq:NN  \UnusedTemplateKeys \c_empty_tl
          }
          {
            \keys_set_known:noN { template / #1 / #2 } {#3}
                                \UnusedTemplateKeys
          }
      }
  }
```

(*End of definition for* `\SetKnownTemplateKeys`*. This function is documented on page* ??.)

`\SetTemplateKeys` Same kind of extension for `\SetTemplateKeys`:

```
\cs_set_protected:Npn \SetTemplateKeys #1#2#3 {
    \SetKnownTemplateKeys {#1}{#2}{#3}
%FMi
    \tl_if_empty:NF \UnusedTemplateKeys
       {
         \msg_error:nneo { block } { unknown-keys }
             { current~ template~ instance }  % we don't know which
             \UnusedTemplateKeys
       }
    \tl_set_eq:NN  \UnusedTemplateKeys \c_empty_tl
}
```

(*End of definition for* `\SetTemplateKeys`*. This function is documented on page* ??.)

5.2 Tracing templates and instances

`\template_debug_typeout:n` I guess that tracing macro is needed in several modules, so should become public (or at least kernel).

```
\cs_new_protected:Npn \template_debug_typeout:n { \__template_debug_typeout:n }
```

(*End of definition for* `\template_debug_typeout:n`*. This function is documented on page* ??.)

`\IfTemplateExistsTF` `\IfTemplateExistsT` `\IfTemplateExistsF` `\__template_if_exist:nn` Test if a certain template was defined (so that it can be used in instance declarations.

```
\prg_new_conditional:Npnn \__template_if_exist:nn #1#2 { T, F, TF }
  {
    \cs_if_exist:cTF { \c__template__keytypes_root_tl #1 / #2 }
      \prg_return_true:
      \prg_return_false:
  }
\cs_new:Npn \IfTemplateExistsTF #1#2
  { \__template_if_template_exist:nnTF {#1} {#2} }
\cs_new:Npn \IfTemplateExistsT #1#2
  { \__template_if_template_exist:nnT {#1} {#2} }
```

```
\cs_new:Npn \IfTemplateExistsF #1#2
  { \__template_if_template_exist:nnF {#1} {#2} }
```

(*End of definition for* `\IfTemplateExistsTF` *and others. These functions are documented on page* **??**.)

```
\ExplSyntaxOff
```

⟨/package⟩

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

U