beamer named overlay ranges with beanover

Jérôme Laurens

version 2022.4 2022/10/01

Abstract

This package allows the management of multiple slide ranges in `beamer` documents. Slide ranges are very handy both during edition and to manage complex and variable overlay specifications.

Contents

1 Minimal example

The document below is a contrived example to show how the `beamer` overlay specifications have been extended.

```
1  \documentclass {beamer}
2  \RequirePackage {beanover}
3  \begin{document}
4  \begin{frame}
5  {\Large Frame \insertframenumber}
6  {\Large Slide \insertslidenumber}
7  \Beanover{
8  A = 1:2,
9  B = A.next:3,
10 C = B.next,
11 }
12 \visible<?(A.1)> {Only on slide 1}\\
13 \visible<?(B.1)-?(B.last)> {Only on slide 3 to 5}\\
14 \visible<?(C.1)> {Only on slide 6}\\
15 \visible<?(A.2)> {Only on slide 2}\\
16 \visible<?(B.2)-?(B.last)> {Only on slide 4 to 5}\\
17 \visible<?(C.2)> {Only on slide 7}\\
18 \visible<?(A.3)-> {From slide 3}\\
19 \visible<?(B.3)-?(B.last)> {Only on slide 5}\\
20 \visible<?(C.3)> {Only on slide 8}\\
21 \end{frame}
22 \end{document}
```

On line 8, we declare a slide range named 'A', starting at slide 1 and with length 2. On line 12, the new overlay specification `?(A.1)` stands for 1, on line 15, `?(A.2)` stands for 2 and on line 18, `?(A.3)` stands for 3. On line 9, we declare a second slide range named 'B', starting after the 2 slides of 'A' namely 3. Its length is 3 meaning that its last side has number 5, thus each `?(B.last)` is replaced by 5. The next slide after time line 'B' has number 6 which is also the first slide of the third time line due to line 10.

2 What is a named slide range?

Within a frame, there are different slides that appear in turn. The main slide range covers all the slide numbers, from one to the total amount of slides. In general, a slide range is a range of positive integers identified by a unique name. The main practical interest is that time lines may be defined relative to one another. Moreover we can specify overlay specifications based on time lines.

3 Defining named slide ranges

`\Beanover` `\Beanover{`⟨*key-value list*⟩`}`

The keys are the slide ranges names, they must contain no spaces nor dots. When the same key is used multiple times, only the last is taken into account. The possible values are ⟨*start*⟩, ⟨*start*⟩:⟨*length*⟩, ⟨*start*⟩::⟨*end*⟩ or ⟨*start*⟩! where ⟨*start*⟩, ⟨*end*⟩ and ⟨*length*⟩ are algebraic expression involving any named overlay specification when an integer.

4 Named overlay specifications

The named overlay specifications are detailled in the tables below together with their replacement meaning value as beamer standard overlay specification.

syntax	**meaning**
⟨*name*⟩ = {i, $i+1$, $i+2$, ...}	
⟨***name***⟩	i
⟨***name***⟩`.1`	i
⟨***name***⟩`.2`	$i+1$
⟨***name***⟩`.`⟨***integer***⟩	$i + \langle integer \rangle - 1$

In the frame example below, we use the `\BeanoverEval` command for the demonstration. It is mainly used for debugging and testing purposes.

```
\begin{frame} {Frame \insertframenumber} {Slide \insertslidenumber}
\Beanover{
A = 3,
}
\ttfamily
\BeanoverEval(A)   ==3,
\BeanoverEval(A.1) ==3,
\BeanoverEval(A.2) ==4,
\BeanoverEval(A.-1)==1,
\end{frame}
```

For finite time lines, we also have

syntax	**meaning**	**output**	
⟨*name*⟩ = {i, $i+1$, ..., j}			
⟨***name***⟩`.length`	$j - i + 1$	A.length	6
⟨***name***⟩`.last`	j	A.last	8
⟨***name***⟩`.next`	$j+1$	A.next	9
⟨***name***⟩`.range`	i `"-"` j	A.range	3-8

```
\begin{frame} {Frame \insertframenumber} {Slide \insertslidenumber}
\Beanover{
A = 3:6,
}
\ttfamily
\BeanoverEval(A.length) == 6,
\BeanoverEval(A.1)      == 3,
\BeanoverEval(A.2)      == 4,
\BeanoverEval(A.-1)     == 1,
\end{frame}
```

Using these specification on unfinite time lines is unsupported. Finally each time line has a dedicated cursor ⟨***name***⟩`.n` that we can use and increment.

⟨***name***⟩`.n` : use the position of the cursor

⟨***name***⟩`.n+=`⟨***integer***⟩ : advance the cursor by ⟨*integer*⟩ and use the new position

`++`⟨***name***⟩`.n` : advance the cursor by 1 and use the new position

5 ?(...) expressions

beamer defines ⟨*overlay specifications*⟩ included between pointed brackets. Before they are processed by the beamer class, the beanover package scans the ⟨*overlay specifications*⟩ for any occurrence of '?(⟨*queries*⟩)'. Each of them is then evaluated and replaced by its static counterpart. The overall result is finally forwarded to beamer.

The ⟨*queries*⟩ argument is a comma separated list of individual ⟨*query*⟩'s. Each ⟨*query*⟩ may be one of '⟨*start*⟩', '⟨*start*⟩:⟨*length*⟩' or '⟨*start*⟩::⟨*last*⟩', where ⟨*start*⟩, ⟨*length*⟩ and ⟨*end*⟩ both denote algebraic expressions possibly involving named overlay specifications. For example `?(A.next)`, `?(A.last+1)`, `?(A.1+A.length)` give the same result as soon as the slide range named '`A`' has been defined with a length.

```
⟨*package⟩
```

6 Implementation

Identify the internal prefix (LaTeX3 DocStrip convention).

```
⟨@@=beanover⟩
```

6.1 Package declarations

```
\NeedsTeXFormat{LaTeX2e}[2020/01/01]
\ProvidesExplPackage
  {beanover}
  {2022/09/22}
  {0.2}
  {Named overlay specifications for beamer}
```

6.2 Overlay specification

6.2.1 In slide range definitions

\g__beanover_prop

⟨*key*⟩–⟨*value*⟩ property list to store the slide ranges. The keys are ⟨*name*⟩, ⟨*name*⟩`.l` where ⟨*name*⟩ is a slide range identifier

```
\prop_new:N \g__beanover_prop
```

(*End definition for* `\g__beanover_prop`.)

Utility message.

```
\msg_new:nnn { beanover } { :n } { #1 }
```

6.2.2 Defining side ranges

\Beanover

`\Beanover {`⟨*key-value list*⟩`}`

The keys are the slide range names.

```
\NewDocumentCommand \Beanover { m } {
  \group_begin:
  \keyval_parse:NNn \__beanover_error:n \__beanover_parse:nn { #1 }
  \group_end:
  \ignorespaces
}
```

We do not accept key only items, they are managed by `\__beanover_error:n`. ⟨*key–value*⟩ items are parsed by `\__beanover_parse:nn`.

__beanover_error:n

Prints an error message when a key only item is used.

```
\cs_new:Npn \__beanover_error:n #1 {
  \msg_fatal:nnn { beanover } { :n } { Missing~value~for~#1 }
}
```

__beanover_parse:nn

__beanover_parse:nn {⟨*name*⟩} {⟨*definition*⟩}

Auxiliary function called within a group. ⟨*name*⟩ is the slide range name, ⟨*definition*⟩ is the definition. Local variables \l_tmpa_seq and \l_tmpb_seq.

\c__beanover_range_regex

Capture groups:

2: the start of the slide range

3: the second colon

4: the length or the end of the range

```
\regex_const:Nn \c__beanover_range_regex {
  \A \s* ([^:]+?) \s* (?: \: (\:)? \s * ( .*? ) \s* )? \Z
}
```

(*End definition for* \c__beanover_range_regex.)

```
\cs_new:Npn \__beanover_parse:nn #1 #2 {
  \regex_extract_once:NnNTF \c__beanover_key_regex { #1 } \l_tmpa_seq {
```

We got a valid key.

```
    \exp_args:Nx
    \tl_if_empty:nTF { \seq_item:Nn \l_tmpa_seq 2 } {
      \regex_extract_once:NnNTF \c__beanover_range_regex { #2 } \l_tmpb_seq {
        \exp_args:Nx
        \tl_if_empty:nTF { \seq_item:Nn \l_tmpb_seq 3 } {
```

This is not a ⟨*start*⟩::⟨*end*⟩ value.

```
          \exp_args:Neee
          \__beanover_l:nnn
            { #1 }
            { \seq_item:Nn \l_tmpb_seq { 2 } }
            { \seq_item:Nn \l_tmpb_seq { 4 } }
        } {
          \exp_args:Neee
          \__beanover_n:nnn
            { #1 }
            { \seq_item:Nn \l_tmpb_seq { 2 } }
            { \seq_item:Nn \l_tmpb_seq { 4 } }
        }
      } {
        \msg_error:nnn { beanover } { :n } { Invalid~declaration:~#2 }
      }
    } {
```

This is an alias.

```
      \prop_gput:Nnn \g__beanover_prop { #1 } { #2 }
    }
  } {
```

```
    \msg_error:nnn { beanover } { :n } { Invalid~declaration:~#1 }
  }
}
```

__beanover_l:nnn `\__beanover:nnn_l {⟨name⟩} {⟨start⟩} {⟨length⟩}`

Auxiliary function called within a group. The length defaults to 1.

```
\cs_new:Npn \__beanover_l:nnn #1 #2 #3 {
  \prop_gput:Nnn \g__beanover_prop { #1.1 } { #2 }
  \prop_gput:Nnn \g__beanover_prop { #1.l } { #3 }
}
```

__beanover_n:nnn `\__beanover_n:nnn {⟨name⟩} {⟨start⟩} {⟨end⟩}`

Auxiliary function called within a group. The ⟨*end*⟩ defaults to {⟨*start*⟩}.

```
\cs_new:Npn \__beanover_n:nnn #1 #2 #3 {
  \prop_gput:Nnn \g__beanover_prop { #1.1 } { #2 }
  \tl_if_empty:nF { #3 } {
    \prop_gput:Nnn \g__beanover_prop { #1.l } { #3 - #1.0 }
  }
}
```

6.2.3 Scanning named overlay specifications

Patch some beamer command to support `?(...)` instructions in overlay specifications.

\l__beanover_scan_tl Storage for the translated overlay specification, where `?(...)` instructions are replaced by their static counterparts.

```
\tl_new:N \l__beanover_scan_tl
```

(*End definition for* `\l__beanover_scan_tl`.)

Save the original macro `\beamer@masterdecode` and then override it to properly preprocess the argument.

```
\cs_set_eq:NN \__beanover_beamer@masterdecode \beamer@masterdecode
\cs_set:Npn \beamer@masterdecode #1 {
  \group_begin:
  \tl_clear:N \l__beanover_scan_tl
  \__beanover_scan:Nn \l__beanover_scan_tl { #1 }
  \exp_args:NNV
  \group_end:
  \__beanover_beamer@masterdecode \l__beanover_scan_tl
}
```

__beanover_scan:n `\__beanover_scan:Nn ⟨tl variable⟩ {⟨named overlay expression⟩}`

Scan the ⟨*named overlay expression*⟩ argument and feed the ⟨*tl variable*⟩ replacing `?(...)` instructions with their static counterpart. A group is created to use local variables:

`\l__beanover_eval_tl:` is the returned token list that will be appended to ⟨*tl variable*⟩ before "exiting".

\l__beanover_depth_int Store the depth level in parenthesis grouping used when finding the proper closing parenthesis balancing the opening parenthesis that follows immediately a question mark in a `?(...)` instruction.

```
\int_new:N \l__beanover_depth_int
```

(*End definition for* `\l__beanover_depth_int`.)

\l__beanover_query_tl Storage for the overlay query expression to be evaluated.

```
\tl_new:N \l__beanover_query_tl
```

(*End definition for* `\l__beanover_query_tl`.)

\l__beanover_token_seq The ⟨*overlay expression*⟩ is split into the sequence of its tokens.

```
\seq_new:N \l__beanover_token_seq
```

(*End definition for* `\l__beanover_token_seq`.)

\l__beanover_ask_bool Whether a loop may continue. Controls the continuation of the main loop that scans the tokens of the ⟨*named overlay expression*⟩ looking for a question mark.

```
\bool_new:N \l__beanover_ask_bool
```

(*End definition for* `\l__beanover_ask_bool`.)

\l__beanover_query_bool Whether a loop may continue. Controls the continuation of the secondary loop that scans the tokens of the ⟨*overlay expression*⟩ looking for an opening parenthesis follow the question mark. It then controls the loop looking for the balanced closing parenthesis.

```
\bool_new:N \l__beanover_query_bool
```

(*End definition for* `\l__beanover_query_bool`.)

\l__beanover_token_tl Whether a loop may continue.

```
\tl_new:N \l__beanover_token_tl
```

(*End definition for* `\l__beanover_token_tl`.)

```
\cs_new:Npn \__beanover_scan:Nn #1 #2 {
  \group_begin:
```

The integer variable are used for readonly data.

```
  \tl_clear:N \l__beanover_eval_tl
  \int_zero:N \l__beanover_depth_int
```

Explode the ⟨*named overlay expression*⟩ into a list of tokens:

```
  \regex_split:nnN {} { #2 } \l__beanover_token_seq
```

Continue the top level loop to scan for a '?':

```
  \bool_set_true:N \l__beanover_ask_bool
  \bool_while_do:Nn \l__beanover_ask_bool {
```

```
    \seq_pop_left:NN \l__beanover_token_seq \l__beanover_token_tl
    \quark_if_no_value:NTF \l__beanover_token_tl {
```

We reached the end of the sequence (and the token list), we end the loop here.

```
      \bool_set_false:N \l__beanover_ask_bool
    } {
```

`\l__beanover_token_tl` contains a 'normal' token.

```
      \tl_if_eq:NnTF \l__beanover_token_tl { ? } {
```

We found a '?', we first gobble tokens until the next '(', —) whatever they may be. In general, no tokens should be silently ignored.

```
        \bool_set_true:N \l__beanover_query_bool
        \bool_while_do:Nn \l__beanover_query_bool {
```

Get next token.

```
          \seq_pop_left:NN \l__beanover_token_seq \l__beanover_token_tl
          \quark_if_no_value:NTF \l__beanover_token_tl {
```

No opening parenthesis found, raise.

```
            \msg_fatal:nnx { beanover } { :n } {Missing~'('%---)
              ~after~a~?:~#2}
          } {
            \tl_if_eq:NnT \l__beanover_token_tl { ( %)
            } {
```

We found the '`(`' after the '`?`'.Increment the parenthesis depth to 1 (on first passage).

```
              \int_incr:N \l__beanover_depth_int
```

Record the forthcomming content in the `\l__beanover_query_tl` variable,up to the next balancing '`)`'.

```
              \tl_clear:N \l__beanover_query_tl
              \bool_while_do:Nn \l__beanover_query_bool {
```

Get next token.

```
                \seq_pop_left:NN \l__beanover_token_seq \l__beanover_token_tl
                \quark_if_no_value:NTF \l__beanover_token_tl {
```

We reached the end of the sequence and the token list with no closing '`)`'. We raise and end both bool while loops. As recovery we feed `\l__beanover_query_tl` with the missing '`)`'. `\l_@@_depth_int` is 0 whenever `\l_@@_query_bool` is false.

```
                  \msg_error:nnx { beanover } { :n } {Missing~%(---
                    ')':~#2 }
                  \int_do_while:nNnn \l__beanover_depth_int > 1 {
                    \int_decr:N \l__beanover_depth_int
                    \tl_put_right:Nn \l__beanover_query_tl {%(---
                    )}
                  }
                  \int_zero:N \l__beanover_depth_int
                  \bool_set_false:N \l__beanover_query_bool
                  \bool_set_false:N \l__beanover_ask_bool
                } {
                  \tl_if_eq:NnTF \l__beanover_token_tl { ( %---)
                  } {
```

We found a '(', increment the depth and append the token to `\l__beanover_query_tl`.

```
\int_incr:N \l__beanover_depth_int
\tl_put_right:NV \l__beanover_query_tl \l__beanover_token_tl
                  } {
```

This is not a '('.

```
                    \tl_if_eq:NnTF \l__beanover_token_tl { %(
                      )
                    } {
```

We found a ')', decrement the depth.

```
                      \int_decr:N \l__beanover_depth_int
                      \int_compare:nNnTF \l__beanover_depth_int = 0 {
```

The depth level has reached 0: we found our balancing parenthesis of the `?(...)` instruction. We can append the evaluated slide ranges token list to `\l__beanover_eval_tl` and stop the inner loop.

```
  \exp_args:NNNx
  \__beanover_eval:NNn \c_false_bool \l__beanover_eval_tl {
    \l__beanover_query_tl
  }
  \bool_set_false:N \l__beanover_query_bool
                      } {
```

The depth has not yet reached level 0.We append the ')' to `\l__beanover_query_tl` because it is not the end of sequence marker.

```
\tl_put_right:NV \l__beanover_query_tl \l__beanover_token_tl
                      }
```

Above ends the code for a positive depth.

```
                    } {
```

The scanned token is not a '(' nor a ')', we append it as is to `\l__beanover_query_tl`.

```
\tl_put_right:NV \l__beanover_query_tl \l__beanover_token_tl
                    }
                  }
                }
```

Above ends the code for Not a '('

```
              }
            }
```

Above ends the code for: Found the '(' after the '?'

```
          }
```

Above ends the code for not a no value quark.

```
        }
```

Above ends the code for the bool while loop to find the '(' after the '?'.

If we reached the end of the token list, then end both the current loop and its containing loop.

```
        \quark_if_no_value:NT \l__beanover_token_tl {
          \bool_set_false:N \l__beanover_query_bool
          \bool_set_false:N \l__beanover_ask_bool
        }
      } {
```

This is not a '?', append the token to right of `\l__beanover_eval_tl` and continue.

```
        \tl_put_right:NV \l__beanover_eval_tl \l__beanover_token_tl
      }
```

Above ends the code for the bool while loop to find a '(' after the '?'

```
    }
  }
```

Above ends the outer bool while loop to find '?' characters. We can append our result to ⟨*tl variable*⟩

```
  \exp_args:NNNV
  \group_end:
  \tl_put_right:Nn #1 \l__beanover_eval_tl
}
```

Each new frame has its own slide ranges set, we clear the property list on entering a new frame environment.

```
\AddToHook
  { env/beamer@framepauses/before }
  { \prop_gclear:N \g__beanover_prop }
```

6.2.4 Evaluation

`\BeanoverEval` `\BeanoverEval [⟨tl variable⟩] {⟨overlay queries⟩}`

⟨*overlay queries*⟩ is the argument of `?(...)` instructions. This is a comma separated list of single ⟨*overlay query*⟩'s.

This function evaluates the ⟨*overlay queries*⟩ and store the result in the ⟨*tl variable*⟩ when provided or leave the result in the input stream. Forwards to `\__beanover_eval:NNn` within a group. `\l__beanover_eval_tl` is used to store the result.

`\l__beanover_eval_tl` Storage for the evaluated result.

```
\tl_new:N \l__beanover_eval_tl
```

(*End definition for* `\l__beanover_eval_tl`.)

```
\NewExpandableDocumentCommand \BeanoverEval { s o m } {
  \group_begin:
  \tl_clear:N \l__beanover_eval_tl
  \exp_args:Nx \__beanover_eval:NNn {
      \IfBooleanTF { #1 } { \c_true_bool } { \c_false_bool }
    }
    \l__beanover_eval_tl { #3 }
  \IfValueTF { #2 } {
    \exp_args:NNNV
    \group_end:
    \tl_set:Nn #2 \l__beanover_eval_tl
  } {
    \exp_args:NV
    \group_end: \l__beanover_eval_tl
  }
}
```

__beanover_eval:NNn `\__beanover_eval:NNn ⟨bool variable⟩ ⟨tl variable⟩ {⟨overlay queries⟩}`

Evaluates the ⟨*overlay queries*⟩, replacing all the named overlay specifications and integer expressions by their static counterparts, then append the result to the right of the ⟨*tl variable*⟩. If the ⟨*bool variable*⟩ is true then the cursor is not available (more explanation required). This is executed within a local group. Below are local variables and constants.

\l__beanover_query_seq Storage for a sequence of queries.

```
\seq_new:N \l__beanover_query_seq
```

(*End definition for* `\l__beanover_query_seq`.)

\l__beanover_eval_seq Storage of the evaluated result.

```
\seq_new:N \l__beanover_eval_seq
```

(*End definition for* `\l__beanover_eval_seq`.)

\c__beanover_comma_regex Used to parse slide range overlay specifications.

```
\regex_const:Nn \c__beanover_comma_regex { \s* ,  \s* }
```

(*End definition for* `\c__beanover_comma_regex`.)

```
\cs_new:Npn \__beanover_eval:NNn #1 #2 #3 {
  \group_begin:
  \tl_clear:N \l__beanover_eval_tl
  \____beanover_eval:NNn #1 \l__beanover_eval_tl { #3 }
```

In this last step, we evaluate the integer expression and put the result in a variable which content will be copied after the group is closed. We authorize comma separated expressions and colon range expressions as well. We first split the expression around commas, into `\l__beanover_query_seq`.

```
  \exp_args:NNV
  \regex_split:NnN \c__beanover_comma_regex \l__beanover_eval_tl
    \l__beanover_query_seq
```

Then each component is evaluated and the result is stored in `\l__beanover_eval_seq` that we must clear before use.

```
  \seq_clear:N \l__beanover_eval_seq
  \seq_map_tokens:Nn \l__beanover_query_seq {
    \__beanover_query:NNn #1 \l__beanover_eval_seq
  }
```

We have managed all the comma separated components, we collect them back and append them to ⟨*tl variable*⟩.

```
  \tl_set:Nx \l__beanover_eval_tl { \seq_use:Nn \l__beanover_eval_seq , }
  \exp_args:NNNV
  \group_end:
  \tl_put_right:Nn #2 \l__beanover_eval_tl
}
```

__beanover_query:NNn `\__beanover_query:NNn ⟨bool variable⟩ ⟨seq variable⟩ {⟨overlay query⟩}`

Evaluates the single ⟨*overlay query*⟩, which is expected to contain no comma, replacing all the named overlay specifications and integer expressions by their static counterparts, then append the result to the right of the ⟨*seq variable*⟩. If the ⟨*bool variable*⟩ is true then the cursor is not available. This is executed within a local group. Below are local variables and constants.

\l__beanover_match_seq Storage for the match.

```
\seq_new:N \l__beanover_match_seq
```

(*End definition for* `\l__beanover_match_seq`.)

\l__beanover_a_tl Storage for the start of a range.

```
\tl_new:N \l__beanover_a_tl
```

(*End definition for* `\l__beanover_a_tl`.)

\l__beanover_b_tl Storage for the end of a range, or its length.

```
\tl_new:N \l__beanover_b_tl
```

(*End definition for* `\l__beanover_b_tl`.)

\g__beanover_colon_regex Used to parse slide range overlay specifications. Next are the capture groups.

2: ⟨*start*⟩

3: Second colon

4: ⟨*end*⟩ or ⟨*length*⟩

```
\regex_const:Nn \c__beanover_colon_regex {
  \A \s*( [^\:]*? ) \s* \: \s* (\:)? \s* ( [^\:]*? ) \s* \Z
}
```

(*End definition for* `\g__beanover_colon_regex`.)

```
\cs_new:Npn \__beanover_query:NNn #1 #2 #3 {
  \regex_extract_once:NnNTF \c__beanover_colon_regex {
    #3
  } \l__beanover_match_seq {
```

We captured colon syntax ranges: one of ⟨*start*⟩:⟨*length*⟩ or ⟨*start*⟩::⟨*last*⟩. We recover the ⟨*start*⟩ and ⟨*end*⟩ or ⟨*length*⟩ respectively in `\l__beanover_a_tl` and `\l__beanover_b_tl`.

```
    \tl_set:Nx \l__beanover_a_tl  { \seq_item:Nn \l__beanover_match_seq 2 }
    \tl_set:Nx \l__beanover_b_tl  { \seq_item:Nn \l__beanover_match_seq 4 }
    \exp_args:Nx
    \tl_if_empty:nTF { \seq_item:Nn \l__beanover_match_seq 3 } {
```

This is a ⟨*start*⟩:⟨*length*⟩ range,

```
      \tl_if_empty:VT \l__beanover_a_tl {
```

raise when ⟨*start*⟩ is void because we cannot evaluate the last index without knowing the first.

```
        \msg_error:nnn { beanover } { :n } { Missing~range~start:~#1 }
        \tl_set:Nn \l__beanover_a_tl 1
      }
```

When not provided, ⟨*length*⟩ defaults to ∞. If there is a ⟨*length*⟩, evaluate it.

```
      \tl_if_empty:VF \l__beanover_b_tl {
        \tl_set:Nx \l__beanover_b_tl { \fp_to_int:n {
          \l__beanover_a_tl + \l__beanover_b_tl - 1
      } }
    }
  } {
```

This is a ⟨*start*⟩`::`⟨*end*⟩ range, with optional ⟨*start*⟩ and ⟨*end*⟩. If there is ⟨*start*⟩, evaluate it,

```
    \tl_if_empty:VF \l__beanover_a_tl {
      \tl_set:Nx \l__beanover_a_tl {
        \exp_args:NV \fp_to_int:n \l__beanover_a_tl
      }
    }
```

and if there is an ⟨*end*⟩, evaluate it as well.

```
      \tl_if_empty:VF \l__beanover_b_tl {
        \tl_set:Nx \l__beanover_b_tl {
          \exp_args:NV \fp_to_int:n \l__beanover_b_tl
        }
      }
    }
```

We can store the range.

```
    \exp_args:NNx
    \seq_put_right:Nn \l__beanover_eval_seq {
      \l__beanover_a_tl - \l__beanover_b_tl
    }
  } {
```

This is not a colon syntax range: we just evaluate the component and store the result, if any.

```
    \tl_if_empty:nF { #3 } {
      \exp_args:NNx
      \seq_put_right:Nn \l__beanover_eval_seq { \fp_to_int:n { #3 } }
    }
  }
}
```

____beanover_eval:NNn `\____beanover_eval:NNn ⟨bool variable⟩ ⟨tl variable⟩ {⟨integer expression⟩}`

Evaluates the ⟨*integer expression*⟩, replacing all the named specifications by their counterpart then put the result to the right of the ⟨*tl variable*⟩. If the ⟨*boolean variable*⟩ is true then the cursor is not available. Executed within a group. Local variables: `\l__beanover_eval_tl` fo the content of ⟨*tl variable*⟩

\l__beanover_split_seq The sequence of queries and non queries.

```
\seq_new:N \l__beanover_split_seq
```

(*End definition for* `\l__beanover_split_seq`.)

\l__beanover_split_int Is the index of the non queries, before all the catched groups.

```
\int_new:N \l__beanover_split_int
```

(*End definition for* `\l__beanover_split_int`.)

\l__beanover_name_tl Storage for `\l__beanover_split_seq` items that represent names.

```
\tl_new:N \l__beanover_name_tl
```

(*End definition for* `\l__beanover_name_tl`.)

\l__beanover_static_tl Storage for the static values of named slide ranges.

```
\tl_new:N \l__beanover_static_tl
```

(*End definition for* `\l__beanover_static_tl`.)

\l__beanover_group_tl Storage for capture groups.

```
\tl_new:N \l__beanover_group_tl
```

(*End definition for* `\l__beanover_group_tl`.)

\c__beanover_int_regex A decimal integer with an eventual sign.

```
\regex_const:Nn \c__beanover_int_regex {
  (?:[-+]\s*)?[0-9]+
}
```

(*End definition for* `\c__beanover_int_regex`.)

\c__beanover_key_regex The name of a slide range consists of an alphabetical character eventually followed by any alphanumerical character. A leading underscore may be used for aliases. Under development.

```
\regex_const:Nn \c__beanover_id_regex {
  [[:alpha:]][[:alnum:]_]*
}
\regex_const:Nn \c__beanover_key_regex {
  \A (_)? \ur{c__beanover_id_regex} \Z
}
```

(*End definition for* `\c__beanover_key_regex`.)

\c__beanover_split_regex Used to parse slide ranges overlay specifications. Next are the capture groups. Group numbers are 1 based because it is used in splitting contex where only capture groups are considered.

(*End definition for* `\c__beanover_split_regex`.)

```
\regex_const:Nn \c__beanover_split_regex {
  ( ? :
```

1: optional prefix increment `++`

```
        ( \+\+ )? \s*
```

2: ⟨*name*⟩

```
        ( \ur{c__beanover_id_regex} ) \s*
        ( ? :
```

3: the integer after `+=`

```
          \+= \s* ( \ur{c__beanover_int_regex} )
          | ( ? : \. \s* ( ? :
```

4: `length`

```
            (l)ength\b
```

5: `range`

```
          | (r)ange\b
```

6: `last`

```
          | (l)ast\b
```

7: `next`

```
          | (n)ext\b
```

8: the integer after the dot

```
          | ( \ur{c__beanover_int_regex} )
```

9: `reset`

```
          | (r)eset\b
```

10: Unknown attribute:

```
          | (\S*)
        ) \s* ) ) ? )
```

11: Alias

```
        | ( _ \ur{c__beanover_id_regex} )

}
```

```
\cs_new:Npn \____beanover_eval:NNn #1 #2 #3 {
  \group_begin:
  \regex_split:NnN \c__beanover_split_regex { #3 } \l__beanover_split_seq
  \int_set:Nn \l__beanover_split_int { 1 }
  \tl_set:Nx \l__beanover_eval_tl {
    \seq_item:Nn \l__beanover_split_seq { \l__beanover_split_int }
  }
```

The `++` prefix should not be given when postfix attributes are.

`\__beanover_tmpa:n` `\__beanover_tmpa:n {⟨code⟩}`

Helper function defined locally. Execute the ⟨*code*⟩ if the `++` prefix is not catched, "raises" an exception otherwise.

```
  \cs_set:Npn \__beanover_tmpa:n ##1 {
    \exp_args:Nx
    \tl_if_empty:nTF {
      \seq_item:Nn \l__beanover_split_seq { \l__beanover_split_int + 1 }
    } {
      ##1
    } {
      \msg_fatal:nnn { beanover } { :n } {
        Unexpected~beanover~specification~(prefix):~ #3
      }
    }
  }
```

`\__beanover:nTF` `\__beanover:nTF {⟨capture group number⟩} {⟨empty code⟩} {⟨non empty code⟩}`

Helper function to locally set the `\l_@@_group_tl` variable to the captured group ⟨*capture group number*⟩ and branch.

```
  \cs_set:Npn \__beanover:nTF ##1 ##2 ##3 {
    \tl_set:Nx \l__beanover_group_tl {
      \seq_item:Nn \l__beanover_split_seq { \l__beanover_split_int + ##1 }
    }
    \tl_if_empty:NTF \l__beanover_group_tl { ##2 } { ##3 }
  }
```

Main loop.

```
  \int_while_do:nNnn { \l__beanover_split_int } < {
    \seq_count:N \l__beanover_split_seq
  } {
```

We first manage the aliases.

```
    \tl_set:Nx \l__beanover_name_tl {
      \seq_item:Nn \l__beanover_split_seq { \l__beanover_split_int + 11 }
    }
    \tl_if_empty:NTF \l__beanover_name_tl {
      \tl_set:Nx \l__beanover_name_tl {
        \seq_item:Nn \l__beanover_split_seq { \l__beanover_split_int + 2 }
      }
      \__beanover:nTF { 3 } { % +=
        \__beanover:nTF { 4 } { % length
          \__beanover:nTF { 5 } { % range
```

```
            \__beanover:nTF { 6 } { % last
              \__beanover:nTF { 7 } { % next
                \__beanover:nTF { 8 } { % .123
                  \__beanover:nTF { 9 } { % .reset
                    \__beanover:nTF { 10 } { % .ERROR
```

Case ⟨*name*⟩

```
\__beanover:nTF { 1 } { % ++
```

Case ++⟨*name*⟩

```
  \__beanover_tmpa:n {
    \bool_if:NT #1 {
      \msg_fatal:nnn { beanover } { :n } {
        No~\l__beanover_name_tl~cursor~available~inside~\cs{Beanover}:~#3
      }
    }
    \exp_args:NNV
    \__beanover_cursor:Nn \l__beanover_static_tl \l__beanover_name_tl
  }
} {
```

Case ++⟨*name*⟩

```
  \bool_if:NT #1 {
    \msg_fatal:nnn { beanover } { :n } {
      No~\l__beanover_name_tl~cursor~available~inside~\cs{Beanover}:~#3
    }
  }
  \exp_args:NNV \__beanover_incr:Nnn \l__beanover_eval_tl \l__beanover_name_tl 1
}
                        } {
```

Case ⟨*name*⟩.UNKNOWN

```
\msg_fatal:nnn { beanover } { :n } { Unknwon~attribute:~#3 }
                        }
                      } {
```

Case ⟨*name*⟩.reset

```
\bool_if:NT #1 {
  \msg_fatal:nnn { beanover } { :n } {
    No~\l__beanover_name_tl~cursor~available~inside~\cs{Beanover}:~#3
  }
}
\exp_args:NnV
\__beanover_reset:nn { 0 } \l__beanover_name_tl
\__beanover:nTF { 1 } { % ++
```

Case ⟨*name*⟩

```
%^^A  \typeout{DEBUG~NAME~1}
  \exp_args:NNV
  \__beanover_cursor:Nn \l__beanover_eval_tl \l__beanover_name_tl
%^^A  \typeout{DEBUG~NAME~LAST}
} {
```

Case ++⟨*name*⟩

```
  \exp_args:NNV \__beanover_incr:Nnn \l__beanover_eval_tl \l__beanover_name_tl 1
}
                    }
                  } {
```

Case ⟨*name*⟩.⟨*integer*⟩

```
\__beanover_tmpa:n {
  \group_begin:
  \tl_clear:N \l__beanover_eval_tl
  \exp_args:NNV \__beanover_start:Nn \l__beanover_eval_tl \l__beanover_name_tl
  \tl_put_right:Nn \l__beanover_eval_tl { + ( \l__beanover_group_tl ) - 1 }
  \exp_args:NNNx
  \group_end:
  \tl_put_right:Nn \l__beanover_eval_tl {
    \fp_to_int:n \l__beanover_eval_tl
  }
}
                  }
                } {
```

Case ⟨*name*⟩.next

```
\__beanover_tmpa:n {
  \exp_args:NNV \__beanover_next:Nn \l__beanover_eval_tl \l__beanover_name_tl
}
                }
              } {
```

Case ⟨*name*⟩.last

```
\__beanover_tmpa:n {
  \exp_args:NNV \__beanover_last:Nn \l__beanover_eval_tl \l__beanover_name_tl
}
              }
            } {
```

Case ⟨*name*⟩.range

```
\__beanover_tmpa:n {
  \bool_if:NT #1 {
    \msg_fatal:nnn { beanover } { :n } {
      No~#3.range available:~
        \seq_item:Nn \l__beanover_split_seq { \l__beanover_split_int + 1 }
    }
  }
  \exp_args:NNV \__beanover_start:Nn \l__beanover_eval_tl \l__beanover_name_tl
  \tl_put_right:Nn \l__beanover_eval_tl { :: }
  \exp_args:NNV \__beanover_last:Nn \l__beanover_eval_tl \l__beanover_name_tl
}
          }
        } {
```

Case ⟨*name*⟩.length

```
\__beanover_tmpa:n {
  \exp_args:NNV \__beanover_length:Nn \l__beanover_eval_tl \l__beanover_name_tl
}
        }
      } {
```

Case ⟨*name*⟩+=⟨*integer*⟩

```
\bool_if:NT #1 {
  \msg_fatal:nnn { beanover } { :n } {
    No~\g_tmpb_tl~cursor~available~inside~\cs{Beanover}:~#3
  }
```

```
}
\exp_args:NNVV
\__beanover_incr:Nnn \l__beanover_eval_tl
  \l__beanover_name_tl \l__beanover_group_tl
      }
    } {
```

This was an alias, go recursive.

```
      \exp_args:NNV
      \prop_if_in:NnTF \g__beanover_prop \l__beanover_name_tl {
        \tl_set:Nx \g_tmpb_tl {
          \exp_args:NNV
          \prop_item:Nn \g__beanover_prop \l__beanover_name_tl
        }
        \tl_if_empty:NT \g_tmpb_tl {
          \tl_set:Nn \g_tmpb_tl { :: }
        }
      } {
        \exp_args:Nnnx
        \msg_error:nnn { beanover } { :n } {
          Unknown~ alias:~\tl_use:N \g_tmpb_tl\space(in~#3)
        }
        \tl_set:Nn \l__beanover_name_tl { :: }
      }
      \exp_args:NNNx
      \____beanover_eval:NNn \c_false_bool \l__beanover_eval_tl \g_tmpb_tl
    }
    \int_add:Nn \l__beanover_split_int { 12 }
    \tl_put_right:Nx \l__beanover_eval_tl {
      \seq_item:Nn \l__beanover_split_seq { \l__beanover_split_int }
    }
  }
  \exp_args:NNNV
  \group_end:
  \tl_put_right:Nn #2 \l__beanover_eval_tl
}
```

__beanover_start:Nn `\__beanover_start:Nn ⟨tl variable⟩ {⟨name⟩}`

Append the start of the ⟨*name*⟩ slide range to the ⟨*tl variable*⟩ with `\____beanover_eval:NNn`. Cache the result.

```
\cs_new:Npn \__beanover_start:Nn #1 #2 {
  \prop_if_in:NnTF \g__beanover_prop { #2.A } {
    \tl_put_right:Nx #1 {
      \prop_item:Nn \g__beanover_prop { #2.A }
    }
  } {
    \group_begin:
    \tl_clear:N \l__beanover_eval_tl
    \prop_if_in:NnTF \g__beanover_prop { #2.0 } {
      \exp_args:NNNx
      \__beanover_eval:NNn \c_true_bool \l__beanover_eval_tl {
        \prop_item:Nn \g__beanover_prop { #2.0 } + 0
      }
    } {
      \exp_args:NNNx
      \__beanover_eval:NNn \c_false_bool \l__beanover_eval_tl {
        \prop_item:Nn \g__beanover_prop { #2.1 } + 0
      }
    }
    \prop_gput:NnV \g__beanover_prop { #2.A }\l__beanover_eval_tl
    \exp_args:NNNV
    \group_end:
    \tl_put_right:Nn #1 \l__beanover_eval_tl
  }
}
```

__beanover_length:n*TF* `\__beanover_length:nTF {⟨name⟩} {⟨true code⟩} {⟨false code⟩}`

Tests whether the ⟨*name*⟩ slide range has a length.

```
\prg_new_protected_conditional:Npnn \__beanover_length:n #1 { TF } {
  \prop_has_item:NnTF \g__beanover_prop { #1 } {
    \prg_return_true
  } {
    \prg_return_false
  }
}
```

__beanover_length:Nn `\__beanover_length:Nn ⟨tl variable⟩ {⟨name⟩}`

Append the length of the ⟨*name*⟩ slide range to ⟨*tl variable*⟩

```
\cs_new:Npn \__beanover_length:Nn #1 #2 {
  \prop_if_in:NnTF \g__beanover_prop { #2.L } {
    \tl_put_right:Nx #1 { \prop_item:Nn \g__beanover_prop { #2.L } }
  } {
    \__beanover_length:nTF { #2 } {
      \group_begin:
      \tl_clear:N \l__beanover_eval_tl
      \exp_args:NNNx
      \__beanover_eval:NNn \c_true_bool \l__beanover_eval_tl {
```

```
        \prop_item:Nn \g__beanover_prop { #2.l } + 0
      }
      \tl_set:Nx \l__beanover_eval_tl {
        \exp_args:NV \fp_to_int:n \l__beanover_eval_tl
      }
      \prop_gput:NnV \g__beanover_prop { #2.L } \l__beanover_eval_tl
      \exp_args:NNNV
      \group_end:
      \tl_put_right:Nn #1 \l__beanover_eval_tl
    } {
      \msg_error:nnn { beanover } { :n } { No~length~given:~#2 }
      \tl_put_right:Nn #1 { 0 }
    }
  }
}
```

__beanover_next:Nn `\__beanover_next:Nn ⟨tl variable⟩ {⟨name⟩}`

Append the index after the ⟨*name*⟩ slide range to the ⟨*tl variable*⟩.

```
\cs_new:Npn \__beanover_next:Nn #1 #2 {
  \prop_if_in:NnTF \g__beanover_prop { #2.N } {
    \tl_put_right:Nx #1 {
      \prop_item:Nn \g__beanover_prop { #2.N }
    }
  } {
    \__beanover_length:nTF { #2 } {
      \group_begin:
      \tl_clear:N \l__beanover_eval_tl
      \__beanover_start:Nn \l__beanover_eval_tl { #2 }
      \tl_put_right:Nn \l__beanover_eval_tl { + }
      \__beanover_length:Nn \l__beanover_eval_tl { #2 }
      \exp_args:NNNV
      \__beanover_eval:NNn \c_true_bool \l__beanover_eval_tl \l__beanover_eval_tl
      \tl_set:Nx \l__beanover_eval_tl {
        \exp_args:NV \fp_to_int:n \l__beanover_eval_tl
      }
      \prop_gput:NnV \g__beanover_prop { #2.N } \l__beanover_eval_tl
      \exp_args:NNNV
      \group_end:
      \tl_put_right:Nn #1 \l__beanover_eval_tl
    } {
      \msg_error:nnn { beanover } { :n } { No~length~given:~#2 }
      \__beanover_start:Nn #1 { #2 }
    }
  }
}
```

__beanover_last:Nn `\__beanover_last:Nn ⟨tl variable⟩ {⟨name⟩}`

```
\cs_new:Npn \__beanover_last:Nn #1 #2 {
  \prop_if_in:NnTF \g__beanover_prop { #2.Z } {
    \tl_put_right:Nx #1 {
```

```
      \prop_item:Nn \g__beanover_prop { #2.Z }
    }
  } {
    \__beanover_length:nTF { #2 } {
      \group_begin:
      \tl_clear:N \l__beanover_eval_tl
      \__beanover_next:Nn \l__beanover_eval_tl { #2 }
      \tl_put_right:Nn \l__beanover_eval_tl { - 1 }
      \tl_set:Nx \l__beanover_eval_tl {
        \exp_args:NV \fp_to_int:n \l__beanover_eval_tl
      }
      \prop_gput:NnV \g__beanover_prop { #2.Z } \l__beanover_eval_tl
      \exp_args:NNNV
      \group_end:
      \tl_put_right:Nn #1 \l__beanover_eval_tl
    } {
      \msg_error:nnn { beanover } { :n } { No~length~given:~#2 }
      \__beanover_start:Nn #1 { #2 }
    }
  }
}
```

__beanover_cursor:Nn

`\__beanover_cursor:Nn` ⟨*tl variable*⟩ {⟨*name*⟩}

Append the value of the cursor associated to the {⟨*name*⟩} slide range to the right of ⟨*tl variable*⟩.

```
\cs_new:Npn \__beanover_cursor:Nn #1 #2 {
  \group_begin:
  \prop_get:NnNTF \g__beanover_prop { #2 } \l__beanover_eval_tl {
    \tl_clear:N \l__beanover_a_tl
    \__beanover_start:Nn \l__beanover_a_tl {#2}
    \int_compare:nNnT { \l__beanover_eval_tl } < { \l__beanover_a_tl } {
      \tl_set_eq:NN \l__beanover_eval_tl \l__beanover_a_tl
    }
```

Not too low.

```
  } {
    \tl_clear:N \l__beanover_eval_tl
    \__beanover_start:Nn \l__beanover_eval_tl {#2}
    \prop_gput:NnV \g__beanover_prop { #2 } \l__beanover_eval_tl
  }
```

If there is a length, use it to bound the result from above.

```
  \__beanover_length:nTF { #2 } {
    \tl_clear:N \l__beanover_a_tl
    \__beanover_last:Nn \l__beanover_a_tl {#2}
    \int_compare:nNnF { \l__beanover_eval_tl } > {
      \l__beanover_a_tl
    } {
      \tl_set_eq:NN \l__beanover_eval_tl \l__beanover_a_tl
    }
  }
  \exp_args:NNNV
  \group_end:
```

```
  \tl_put_right:Nn #1 \l__beanover_eval_tl
}
```

__beanover_incr:Nnn `\__beanover_incr:Nnn ⟨tl variable⟩ {⟨name⟩} {⟨offset⟩}`

Increment the cursor position accordingly. The result must lay in the declared range.

```
\cs_new:Npn \__beanover_incr:Nnn #1 #2 #3 {
  \group_begin:
  \tl_clear:N \l__beanover_eval_tl
  \__beanover_cursor:Nn \l__beanover_eval_tl { #2 }
  \exp_args:NNx
  \__beanover_eval:Nn \l__beanover_eval_tl { \l__beanover_eval_tl + ( #3 ) }
  \prop_gput:NnV \g__beanover_prop {#2} \l__beanover_eval_tl
  \exp_args:NNNV
  \group_end:
  \tl_put_right:Nn #1 \l__beanover_eval_tl
}
```

6.2.5 Reseting slide ranges

\BeanoverReset `\BeanoverReset {⟨Slide range name⟩}`

```
\NewDocumentCommand \BeanoverReset { O{0} m } {
  \__beanover_reset:nn { #1 } { #2 }
  \ignorespaces
}
```

Forwards to `\__beanover_reset:nn`.

__beanover_reset:nn `\__beanover_reset:nn {⟨start value⟩} {⟨slide range name⟩}`

Reset the cursor to its start value.

```
\cs_new:Npn \__beanover_reset:nn #1 #2 {
  \prop_if_in:NnTF \g__beanover_prop { #2.A } {
    \prop_gremove:Nn \g__beanover_prop { #2.A }
    \prop_gremove:Nn \g__beanover_prop { #2 }
    \prop_gput:Nnn \g__beanover_prop { #2.0 } { #1 }
  } {
    \prop_if_in:NnT \g__beanover_prop { #2.1 } {
      \prop_gremove:Nn \g__beanover_prop { #2 }
      \prop_gput:Nnn \g__beanover_prop { #2.0 } { #1 }
    }
  }
}

\makeatother
\ExplSyntaxOff
```

⟨/package⟩