beamer named overlay ranges with beanover

Jérôme Laurens

? ?

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, $i+1$, $i+2, \ldots$}	
⟨***name***⟩**.1**	i
⟨***name***⟩**.2**	$i+1$
⟨***name***⟩**.**⟨***integer***⟩	$i+$⟨*integer*⟩-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.1) ==3,
\BeanoverEval(A.2) ==4,
\BeanoverEval(A.-1)==1,
\end{frame}
```

For finite time lines, we also have

syntax	meaning	output	
⟨*name*⟩ = {i, $i+1, \ldots,$ 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***⟩ : 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}
  {2022/10/05}
  {0.2}
  {Named overlay specifications for beamer}
```

6.2 Local variables

We make heavy use of local variables and function scopes. Many functions are executed within a TeX group, which ensures no name collision with the caller stack. In that case, variables need not follow exactly the LaTeX3 naming convention: we do not specialize with the module name.

```
\bgroup_begin:
\tl_clear_new:N  \l_a_tl
\tl_clear_new:N  \l_b_tl
\tl_clear_new:N  \l_ans_tl
\seq_clear_new:N \l_ans_seq
\seq_clear_new:N \l_match_seq
\seq_clear_new:N \l_token_seq
\int_zero_new:N  \l_split_int
\seq_clear_new:N \l_split_seq
\int_zero_new:N  \l_depth_int
\tl_clear_new:N  \l_name_tl
\tl_clear_new:N  \l_group_tl
\tl_clear_new:N  \l_query_tl
\seq_clear_new:N \l_query_seq
\bgroup_end:
```

6.3 Overlay specification

6.3.1 In slide range definitions

\g__beanover_prop ⟨*key*⟩–⟨*value*⟩ property list to store the slide ranges. The basic keys are, assuming ⟨*name*⟩ is a slide range identifier,

⟨***name***⟩`.1` for the start index

⟨***name***⟩`.l` for the length when provided

⟨***name***⟩`.n` for the cursor value, when used

⟨***name***⟩`.c` for initial value of the cursor (when reset)

Other keys are eventually used to cache results when some attributes are defined from other slide ranges.

⟨***name***⟩`.A` for the cached start index

⟨***name***⟩`.L` for the cached length

And in case a length has been given

⟨***name***⟩`.N` for the cached next index

⟨***name***⟩`.Z` for the cached last index

We definitely use the fact that ⟨*name*⟩ contains no '.' character.

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

(*End definition for* \g__beanover_prop.)

Utility message.

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

6.3.2 Defining named slide ranges

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

The keys are the slide range names. We do not accept key only items, they are managed by `\__beanover_error:n`. ⟨*key–value*⟩ items are parsed by `\__beanover_parse:nn`. A group is open.

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

`\__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.

\l_match_seq Local storage for the match result.

(*End definition for* \l_match_seq*. This variable is documented on page* **??**.)

\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_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_match_seq {
```

We got a valid key.

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

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

```
          \exp_args:Neee
          \__beanover_l:nnn
            { #1 }
            { \seq_item:Nn \l_match_seq { 2 } }
            { \seq_item:Nn \l_match_seq { 4 } }
        } {
          \exp_args:Neee
          \__beanover_n:nnn
            { #1 }
            { \seq_item:Nn \l_match_seq { 2 } }
            { \seq_item:Nn \l_match_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_l:nnn {⟨*name*⟩} {⟨*start*⟩} {⟨*length*⟩}

Auxiliary function called within a group. The ⟨*length*⟩ may be empty. Set the keys {⟨*name*⟩}.1 and eventually {⟨*name*⟩}.l.

```
\cs_new:Npn \__beanover_l: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 }
  }
}
```

__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 }
  }
}
```

6.3.3 Scanning named overlay specifications

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

\beamer@masterdecode

\beamer@masterdecode {⟨*overlay specification*⟩}

Preprocess ⟨*overlay specification*⟩ before beamer uses it.

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

(*End definition for* \l_ans_tl*. This variable is documented on page* **??**.)

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_ans_tl
  \__beanover_scan:Nn \l_ans_tl { #1 }
  \exp_args:NNV
  \group_end:
  \__beanover_beamer@masterdecode \l_ans_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 by their static counterpart with help from `\__beanover_eval:Nn`. A group is created to use local variables:

`\l_ans_tl`: is the token list that will be appended to ⟨*tl variable*⟩ on return.

`\l_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.

(*End definition for* `\l_depth_int`*. This variable is documented on page* **??**.)

`\l_query_tl` Storage for the overlay query expression to be evaluated.

(*End definition for* `\l_query_tl`*. This variable is documented on page* **??**.)

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

(*End definition for* `\l_token_seq`*. This variable is documented on page* **??**.)

`\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_token_tl` Storage for just one token.

(*End definition for* `\l_token_tl`*. This variable is documented on page* **??**.)

```
\cs_new:Npn \__beanover_scan:Nn #1 #2 {
  \group_begin:
  \tl_clear:N \l_ans_tl
  \int_zero:N \l_depth_int
  \seq_clear:N \l_token_seq
```

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

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

Run 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_token_seq \l_token_tl
    \quark_if_no_value:NTF \l_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_token_tl` contains a ‘normal’ token.

```
      \tl_if_eq:NnTF \l_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_token_seq \l_token_tl
          \quark_if_no_value:NTF \l_token_tl {
```

No opening parenthesis found, raise.

```
            \msg_fatal:nnx { __beanover } { :n } {Missing~’(’%---)
              ~after~a~?:~#2}
          } {
            \tl_if_eq:NnT \l_token_tl { ( %)
            } {
```

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

```
              \int_incr:N \l_depth_int
```

Record the forthcomming content in the `\l_query_tl` variable, up to the next balancing ‘)‘.

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

Get next token.

```
                \seq_pop_left:NN \l_token_seq \l_token_tl
                \quark_if_no_value:NTF \l_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_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_depth_int > 1 {
                    \int_decr:N \l_depth_int
                    \tl_put_right:Nn \l_query_tl {%(---
                    )}
                  }
                  \int_zero:N \l_depth_int
                  \bool_set_false:N \l__beanover_query_bool
                  \bool_set_false:N \l__beanover_ask_bool
                } {
                  \tl_if_eq:NnTF \l_token_tl { ( %---)
                  } {
```

We found a ‘`(`’, increment the depth and append the token to `\l_query_tl`.

```
                  \int_incr:N \l_depth_int
                  \tl_put_right:NV \l_query_tl \l_token_tl
                } {
```

This is not a ‘`(`’.

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

We found a ‘`)`’, decrement the depth.

```
                    \int_decr:N \l_depth_int
                    \int_compare:nNnTF \l_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_ans_tl` and stop the inner loop.

```
\exp_args:NNNV
\__beanover_eval:NNn \c_false_bool \l_ans_tl \l_query_tl
\bool_set_false:N \l__beanover_query_bool
                    } {
```

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

```
                      \tl_put_right:NV \l_query_tl \l_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_query_tl`.

```
                    \tl_put_right:NV \l_query_tl \l_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_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_ans_tl` and continue.

```
      \tl_put_right:NV \l_ans_tl \l_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_ans_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 }
```

6.3.4 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_ans_tl` is used to store the result.

```
\NewExpandableDocumentCommand \BeanoverEval { s o m } {
  \group_begin:
  \tl_clear:N \l_ans_tl
  \exp_args:Nx \__beanover_eval:NNn {
      \IfBooleanTF { #1 } { \c_true_bool } { \c_false_bool }
    }
    \l_ans_tl { #3 }
  \IfValueTF { #2 } {
    \exp_args:NNNV
    \group_end:
    \tl_set:Nn #2 \l_ans_tl
  } {
    \exp_args:NV
    \group_end: \l_ans_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_query_seq Storage for a sequence of queries.

(*End definition for* \l_query_seq. *This variable is documented on page* **??**.)

\l_ans_seq Storage of the evaluated result.

(*End definition for* \l_ans_seq. *This variable is documented on page* **??**.)

\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.)
No other variable is used.

\c__beanover_eval_regex Used to parse slide range overlay specifications.

```
\regex_const:Nn \c__beanover_eval_regex { \s* ( ?: (,) | (:) | (::) ) \s* }
```

(*End definition for* \c__beanover_eval_regex.)

```
\cs_new:Npn \__beanover_eval:NNn #1 #2 #3 {
  \group_begin:
  \regex_split:NnN \c__beanover_eval_regex { #3 } \l_split_seq
  \int_zero:N \l_split_int

}
```

```
\cs_new:Npn \__beanover_eval_a:NNn #1 #2 #3 {
  \group_begin:
```

Local variables declaration

```
  \tl_clear:N  \l_a_tl
  \tl_clear:N  \l_b_tl
  \tl_clear:N  \l_ans_tl
  \seq_clear:N \l_ans_seq
  \seq_clear:N \l_query_seq
```

In this main evaluation 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 ⟨*start*⟩::⟨*end*⟩ range expressions as well. We first split the expression around commas, into \l_query_seq.

```
  \__beanover_eval_static:NNn #1 \l_ans_tl { #3 }
  \exp_args:NNV
  \regex_split:NnN \c__beanover_comma_regex \l_ans_tl \l_query_seq
```

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

```
  \seq_map_tokens:Nn \l_query_seq {
    \__beanover_eval_query:NNn #1 \l_ans_seq
  }
```

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

```
  \exp_args:NNNx
  \group_end:
  \tl_put_right:Nn #2 { \seq_use:Nn \l_ans_seq , }
}
```

__beanover_eval_query:NNn

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

Evaluates the single ⟨*overlay query*⟩, which is expected to contain no comma. Replaces all the named overlay specifications by their static counterparts, make the computation then append the result to the right of the ⟨*seq variable*⟩. Ranges are supported with the colon syntax. 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_a_tl Storage for the start of a range.

(*End definition for* \l_a_tl*. This variable is documented on page* ??*.*)

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

(*End definition for* \l_b_tl*. This variable is documented on page* ??*.*)

\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_eval_query:NNn #1 #2 #3 {
  \regex_extract_once:NnNTF \c__beanover_colon_regex {
    #3
  } \l_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_a_tl and \l_b_tl.

```
    \tl_set:Nx \l_a_tl  { \seq_item:Nn \l_match_seq 2 }
    \tl_set:Nx \l_b_tl  { \seq_item:Nn \l_match_seq 4 }
    \exp_args:Nx
    \tl_if_empty:nTF { \seq_item:Nn \l_match_seq 3 } {
```

This is a ⟨`start`⟩`:`⟨`length`⟩ range,

```
      \tl_if_empty:VT \l_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_a_tl 1
      }
```

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

```
   \tl_if_empty:VF \l_b_tl {
    \tl_set:Nx \l_b_tl { \fp_to_int:n {
     \l_a_tl + \l_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_a_tl {
   \tl_set:Nx \l_a_tl {
    \exp_args:NV \fp_to_int:n \l_a_tl
   }
  }
```

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

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

We can store the standard beamer range.

```
  \exp_args:NNx
  \seq_put_right:Nn \l_ans_seq {
   \l_a_tl - \l_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_seq { \fp_to_int:n { #3 } }
  }
 }
}
```

`\__beanover_eval_static:NNn` `\__beanover_eval_static: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 (useful when used from `\Beanover`). Executed within a group. Local variables: `\l_ans_tl` for the content of ⟨*tl variable*⟩

`\l_split_seq` The sequence of queries and non queries.

(*End definition for* `\l_split_seq`*. This variable is documented on page* **??**.)

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

(*End definition for* `\l_split_int`*. This variable is documented on page* **??**.)

`\l_name_tl` Storage for `\l_split_seq` items that represent names.

```
\tl_new:N \l_name_tl
```

(*End definition for* `\l_name_tl`*. This variable is documented on page* **??**.)

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

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

`\l_group_tl` Storage for capture groups.

(*End definition for* `\l_group_tl`*. This variable is documented on page* **??**.)

`\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_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 {
  \s* ( ? :
```

1: optional prefix increment `++`

2: ⟨*name*⟩ of a cursor

```
          ( \+\+ )? ( \ur{c__beanover_id_regex} ) \b
```

3: ⟨*name*⟩ of a cursor

4: the integer after `+=`

```
        | ( \ur{c__beanover_id_regex} ) \s*
            \+= \s* ( \ur{c__beanover_int_regex} )
```

5: ⟨*name*⟩ of a slide range followed by an attribute.

```
       | ( \ur{c__beanover_id_regex} ) \.
         ( ? :
```

6: `length`

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

7: `range`

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

8: `last`

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

9: `next`

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

10: the integer after the dot

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

11: `reset`

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

12: `UNKNOWN`

```
           | ( \S+ )
         )
```

13: Alias

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

```
  ) \s*
}
\cs_new:Npn \__beanover_eval_static:NNn #1 #2 #3 {
  \group_begin:
```

Local variables:

```
  \tl_clear:N  \l_ans_tl
  \int_zero:N  \l_split_int
  \seq_clear:N \l_split_seq
  \tl_clear:N  \l_name_tl
  \tl_clear:N  \l_group_tl
  \tl_clear:N  \l_a_tl
```

Implementation:

```
\regex_split:NnN \c__beanover_split_regex { #3 } \l_split_seq
\int_set:Nn \l_split_int { 1 }
\tl_set:Nx \l_ans_tl { \seq_item:Nn \l_split_seq { \l_split_int } }
```

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

`\guard:n` `\__beanover_a:n {⟨code⟩}`

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

```
\cs_set:Npn \guard:n ##1 {
  \exp_args:Nx
  \tl_if_empty:nTF {
    \seq_item:Nn \l_split_seq { \l_split_int + 1 }
  } {
    ##1
  } {
    \msg_fatal:nnn { __beanover } { :n } {
      Unexpected~beanover~specification~(prefix):~ #3
    }
  }
}
```

`\switch:nTF` `\switch: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 \switch:nNTF ##1 ##2 ##3 ##4 {
  \tl_set:Nx ##2 {
    \seq_item:Nn \l_split_seq { \l_split_int + ##1 }
  }
  \tl_if_empty:NTF ##2 { ##3 } { ##4 }
}
```

Main loop.

```
\int_while_do:nNnn { \l_split_int } < { \seq_count:N \l_split_seq } {
  \switch:nNTF { 2 } \l_name_tl {
    \switch:nNTF { 3 } \l_name_tl {
      \switch:nNTF { 5 } \l_name_tl {
        \switch:nNTF { 13 } \l_name_tl {
```

Unreachable code. PROBLEM WITH ::.

```
        } { % alias
```

Case `_⟨name⟩`. This is an alias, go recursive. Work in progress.

```
\exp_args:NNV
\prop_if_in:NnTF \g__beanover_prop \l_name_tl {
  \tl_set:Nx \l_a_tl {
    \exp_args:NNV
    \prop_item:Nn \g__beanover_prop \l_name_tl
  }
  \tl_if_empty:NT \l_a_tl {
    \tl_set:Nn \l_a_tl { :: }
  }
} {
  \exp_args:Nnnx
  \msg_error:nnn { __beanover } { :n } {
    Unknown~ alias:~\tl_use:N \l_a_tl\space(in~#3)
  }
  \tl_set:Nn \l_a_tl { :: }
}
\exp_args:NNNV
\__beanover_eval_static:NNn \c_false_bool \l_ans_tl \l_a_tl
        }
      } {
```

Case ⟨*name*⟩.⟨*attribute*⟩.

```
        \switch:nNTF { 6 } \l_group_tl { % .length
          \switch:nNTF { 7 } \l_group_tl { % .range
            \switch:nNTF { 8 } \l_group_tl { % .last
              \switch:nNTF { 9 } \l_group_tl { % .next
                \switch:nNTF { 10 } \l_group_tl { % .<integer>
                  \switch:nNTF { 11 } \l_group_tl { % .reset
                    \switch:nNTF { 12 } \l_group_tl { % .UNKNOWN
```

Unreachable code.

```
                    } {
```

Case ⟨*name*⟩.UNKNOWN.

```
\msg_fatal:nnn { __beanover } { :n } { Unknown~attribute~\l_group_tl:~#3 }
                    }
                  } {
```

Case ⟨*name*⟩.reset.

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

Case ⟨*name*⟩.⟨*integer*⟩.

```
\group_begin:
\tl_clear:N \l_ans_tl
\exp_args:NNV \__beanover_start:Nn \l_ans_tl \l_name_tl
\tl_put_right:Nn \l_ans_tl { + ( \l_group_tl ) - 1 }
\exp_args:NNNx
\group_end:
\tl_put_right:Nn \l_ans_tl {
  \fp_to_int:n \l_ans_tl
}
                }
              } {
```

Case ⟨*name*⟩.next.

```
\exp_args:NNV \__beanover_next:Nn \l_ans_tl \l_name_tl
```

```
              }
            } {
```

Case ⟨*name*⟩.last.

```
\exp_args:NNV \__beanover_last:Nn \l_ans_tl \l_name_tl
            }
          } {
```

Case ⟨*name*⟩.range. PROBLEM with ::

```
\bool_if:NT #1 {
  \msg_fatal:nnn { __beanover } { :n } {
    No~\l_name_tl.range available:~#3
  }
}
\exp_args:NNV \__beanover_start:Nn \l_ans_tl \l_name_tl
\tl_put_right:Nn \l_ans_tl { :: }
\exp_args:NNV \__beanover_last:Nn \l_ans_tl \l_name_tl
          }
        } {
```

Case ⟨*name*⟩.length.

```
\exp_args:NNV \__beanover_length:Nn \l_ans_tl \l_name_tl
        }
      }
    } {
      \switch:nNTF { 4 } \l_group_tl { % +=
\msg_fatal:nnn { __beanover } { :n } {
  No~integer~to~increment~\l_name_tl:~#3
}
```

Case ⟨*name*⟩ += ⟨*integer*⟩.

```
      } {
\bool_if:NT #1 {
  \msg_fatal:nnn { __beanover } { :n } {
    No~\l_name_tl~cursor~available~inside~\cs{Beanover}:~#3
  }
}
\exp_args:NNVV
\__beanover_incr:Nnn \l_ans_tl \l_name_tl \l_group_tl
      }
    }
  } {
    \switch:nNTF { 1 } \l_name_tl {
```

Case ⟨*name*⟩.

```
      \bool_if:NT #1 {
        \msg_fatal:nnn { __beanover } { :n } {
          No~\l_name_tl~cursor~available~inside~\cs{Beanover}:~#3
        }
      }
      \exp_args:NNV
      \__beanover_cursor:Nn \l_ans_tl \l_name_tl
    } { % ++ ?
```

Case ++⟨*name*⟩.

```
      \bool_if:NT #1 {
          \msg_fatal:nnn { __beanover } { :n } {
            No~\l_name_tl~cursor~available~inside~\cs{Beanover}:~#3
          }
        }
        \exp_args:NNV
        \__beanover_incr:Nnn \l_ans_tl \l_name_tl 1
      }
    }
    \int_add:Nn \l_split_int { 13 }
    \tl_put_right:Nx \l_ans_tl {
      \seq_item:Nn \l_split_seq { \l_split_int }
    }
  }
  \exp_args:NNNV
  \group_end:
  \tl_put_right:Nn #2 \l_ans_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_static: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_ans_tl
    \prop_if_in:NnTF \g__beanover_prop { #2.c } {
      \exp_args:NNNx
      \__beanover_eval:NNn \c_true_bool \l_ans_tl {
        \prop_item:Nn \g__beanover_prop { #2.c } + 0
      }
    } {
      \exp_args:NNNx
      \__beanover_eval:NNn \c_false_bool \l_ans_tl {
        \prop_item:Nn \g__beanover_prop { #2.1 } + 0
      }
    }
    \prop_gput:NnV \g__beanover_prop { #2.A } \l_ans_tl
    \exp_args:NNNV
    \group_end:
    \tl_put_right:Nn #1 \l_ans_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_ans_tl
      \exp_args:NNNx
      \__beanover_eval:NNn \c_true_bool \l_ans_tl {
        \prop_item:Nn \g__beanover_prop { #2.l } + 0
      }
      \tl_set:Nx \l_ans_tl {
        \exp_args:NV \fp_to_int:n \l_ans_tl
      }
      \prop_gput:NnV \g__beanover_prop { #2.L } \l_ans_tl
      \exp_args:NNNV
      \group_end:
      \tl_put_right:Nn #1 \l_ans_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_ans_tl
      \__beanover_start:Nn \l_ans_tl { #2 }
      \tl_put_right:Nn \l_ans_tl { + }
      \__beanover_length:Nn \l_ans_tl { #2 }
      \tl_clear:N \l_a_tl
      \exp_args:NNNV
      \__beanover_eval:NNn \c_true_bool \l_a_tl \l_ans_tl
      \tl_set:Nx \l_ans_tl {
        \exp_args:NV \fp_to_int:n \l_a_tl
      }
      \prop_gput:NnV \g__beanover_prop { #2.N } \l_ans_tl
      \exp_args:NNNV
      \group_end:
      \tl_put_right:Nn #1 \l_ans_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_ans_tl
      \__beanover_next:Nn \l_ans_tl { #2 }
      \tl_put_right:Nn \l_ans_tl { - 1 }
      \tl_set:Nx \l_ans_tl {
        \exp_args:NV \fp_to_int:n \l_ans_tl
      }
      \prop_gput:NnV \g__beanover_prop { #2.Z } \l_ans_tl
      \exp_args:NNNV
      \group_end:
      \tl_put_right:Nn #1 \l_ans_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_ans_tl {
    \tl_clear:N \l_a_tl
    \__beanover_start:Nn \l_a_tl {#2}
    \int_compare:nNnT { \l_ans_tl } < { \l_a_tl } {
      \tl_set_eq:NN \l_ans_tl \l_a_tl
    }
```

Not too low.

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

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

```
  \__beanover_length:nTF { #2 } {
    \tl_clear:N \l_a_tl
    \__beanover_last:Nn \l_a_tl {#2}
    \int_compare:nNnF { \l_ans_tl } > { \l_a_tl } {
      \tl_set_eq:NN \l_ans_tl \l_a_tl
    }
  }
  \exp_args:NNNV
  \group_end:
\tl_set:Nn #1 \l_ans_tl
}
```

`\__beanover_incr:Nnn`

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

Increment the cursor position accordingly. The result will lay within the declared range.

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

6.3.5 Reseting slide ranges

`\BeanoverReset`

`\BeanoverReset` [⟨*start value*⟩] {⟨*Slide range name*⟩}

```
\NewDocumentCommand \BeanoverReset { O{1} 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 the given ⟨*start value*⟩ which defaults to 1. Clean the cached values also (not usefull).

```
\cs_new:Npn \__beanover_reset:nn #1 #2 {
  \prop_if_in:NnTF \g__beanover_prop { #2.1 } {
    \prop_gremove:Nn \g__beanover_prop { #2 }
    \prop_gremove:Nn \g__beanover_prop { #2.A }
    \prop_gremove:Nn \g__beanover_prop { #2.L }
    \prop_gremove:Nn \g__beanover_prop { #2.N }
    \prop_gremove:Nn \g__beanover_prop { #2.Z }
    \prop_gput:Nnn \g__beanover_prop { #2.c } { #1 }
  } {
    \msg_warning:nnn { __beanover } { :n } { Unknown~name:~#2 }
  }
}
```

```
\makeatother
\ExplSyntaxOff
```

⟨/package⟩