

# beamer named overlay ranges with beanover

Jérôme Laurens

v1.0      2022/10/28

## Abstract

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

## Contents

<b>1</b>	<b>Minimal example</b>	<b>2</b>
<b>2</b>	<b>Named slide lists</b>	<b>2</b>
2.1	Presentation . . . . .	2
2.2	Defining named slide ranges . . . . .	3
2.3	Defining named slide lists . . . . .	3
<b>3</b>	<b>Named overlay specifications</b>	<b>3</b>
3.1	Named slide ranges . . . . .	3
3.2	Named slide lists . . . . .	4
<b>4</b>	<b>Named overlay specifications</b>	<b>4</b>
<b>5</b>	<b>?(...) query expressions</b>	<b>4</b>
<b>6</b>	<b>Implementation</b>	<b>5</b>
6.1	Package declarations . . . . .	5
6.2	Local variables . . . . .	5
6.3	Overlay specification . . . . .	6
6.3.1	In slide range definitions . . . . .	6
6.3.2	Regular expressions . . . . .	8
6.3.3	Defining named slide ranges . . . . .	10
6.3.4	Scanning named overlay specifications . . . . .	13
6.3.5	Evaluation bricks . . . . .	17
6.3.6	Evaluation . . . . .	24
6.3.7	Reseting slide ranges . . . . .	31

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

On line 5, we use the **beanover** key to declare named slide ranges. On line 6, we declare a slide range named ‘A’, starting at slide 1 and with length 2. On line 13, the new overlay specification `?(A.1)` stands for 1, on line 16, `?(A.2)` stands for 2 and on line 19, `?(A.3)` stands for 3. On line 7, 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 slide number is 5, thus each `?(B.last)` is replaced by 5. The next slide number after slide range ‘B’ is 6 which is also the start of the third slide range due to line 8.

## 2 Named slide lists

### 2.1 Presentation

Within a **beamer** frame, there are different slides that appear in turn. The main slide list is a range on integers covering all the slide numbers, from one to the total amount of slides. In general, a slide list is a range of positive integers identified by a unique name. The main practical interest is that such lists may be defined relative to one another, we can even have lists of slide ranges. Finally, we can use these lists to specify **beamer** overlay specifications.

## 2.2 Defining named slide ranges

In order to define named slide ranges, we can either use the `\Beanover` command below inside a `beamer` frame environment, or use the `beanover` option of this environment. The value of the `beanover` option is exactly the argument of the `\Beanover` command. When used, the `\Beanover` command is executed on each frame, whereas the option is executed only once but is more verbose.

---

<code>\Beanover</code>	<code>\Beanover{\langle key--value list \rangle}</code>
------------------------	---

---

The keys are the slide lists names, they are case sensitive and must contain no spaces nor `'` character. When the same key is used multiple times, only the last one is taken into account. Possible values are the *range specifiers*  $\langle first \rangle$ ,  $\langle first \rangle : \langle length \rangle$ ,  $\langle first \rangle : \langle last \rangle$ ,  $: \langle length \rangle : \langle last \rangle$  where  $\langle first \rangle$ ,  $\langle length \rangle$  and  $\langle last \rangle$  are algebraic expression involving any named overlay specification defined next when an integer.

A comma separated list of such specifiers between square brackets is also allowed, which results in a *named slide list*.

The `\Beanover` command is used at the very beginning of the `frame` environment and thus only apply to this frame. It can be used there mutiple times.

## 2.3 Defining named slide lists

The declaration `\Beanover{\langle name \rangle = [\langle spec_1 \rangle, \langle spec_2 \rangle, \dots, \langle spec_n \rangle]}` is a convenient short-cut for `\Beanover{\langle name \rangle.1 = \langle spec_1 \rangle, \langle name \rangle.2 = \langle spec_2 \rangle, \dots, \langle name \rangle.n = \langle spec_n \rangle}`. The rule of the previous section can apply.

# 3 Named overlay specifications

## 3.1 Named slide ranges

For named slide ranges, the named overlay specifications are detailed in the tables below together with their replacement meaning value as `beamer` standard overlay specification.

syntax	meaning
$\langle name \rangle = [i, i + 1, i + 2, \dots]$	
$\langle name \rangle.1$	$i$
$\langle name \rangle.2$	$i + 1$
$\langle name \rangle.\langle integer \rangle$	$i + \langle integer \rangle - 1$

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.1) ==3,
\BeanoverEval(A.2) ==4,
\BeanoverEval(A.-1)==1,
\end{frame}
```

When the slide range has been given a length or an end, like in the frame example below, we also have

syntax	meaning		output
$\langle name \rangle = [i, i + 1, \dots, j]$			
$\langle name \rangle .length$	$j - i + 1$	A.length	6
$\langle name \rangle .last$	$j$	A.last	8
$\langle name \rangle .next$	$j + 1$	A.next	9
$\langle name \rangle .range$	$i \text{ ' ' } j$	A.range	3-8

```
\begin{frame} {Frame \insertframenum} {Slide \insertslidenum}
\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 named slide ranges is unsupported. Finally each named slide range has a dedicated counter  $\langle name \rangle .n$  which is some kind of variable that can be used and incremented.

$\langle name \rangle .n$  : use the position of the counter

$\langle name \rangle .n += \langle integer \rangle$  : advance the counter by  $\langle integer \rangle$  and use the new position

$++\langle name \rangle .n$  : advance the counter by 1 and use the new position

Notice that  $.n$  can generally be omitted.

### 3.2 Named slide lists

The declaration  $\backslash\text{Beanover}\{A=[\langle spec_1 \rangle, \langle spec_2 \rangle, \dots, \langle spec_n \rangle]\}$  is a convenient shortcut for  $\backslash\text{Beanover}\{A.1=\langle spec_1 \rangle, A.2=\langle spec_2 \rangle, \dots, A.n=\langle spec_n \rangle\}$ . The rule of the previous section can apply.

## 4 Named overlay specifications

For named slide ranges, the named overlay specifications are detailed in the tables below together with their replacement meaning value as **beamer** standard overlay specification.

## 5 ?(...) query expressions

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

The  $\langle queries \rangle$  argument is a comma separated list of individual  $\langle query \rangle$ 's of next table. Sometimes, using  $\langle name \rangle.range$  is not allowed as it would lead to an algebraic difference instead of a range.

query	static value	limitation
:	`-'	
::	`-'	
$\langle first \ expr \rangle$	$\langle first \rangle$	
$\langle first \ expr \rangle:$	$\langle first \rangle$ '-'	no $\langle name \rangle.range$
$\langle first \ expr \rangle::$	$\langle first \rangle$ '-'	no $\langle name \rangle.range$
$\langle first \ expr \rangle:\langle length \ expr \rangle$	$\langle first \rangle$ '-' $\langle last \rangle$	no $\langle name \rangle.range$
$\langle first \ expr \rangle:\langle end \ expr \rangle$	$\langle first \rangle$ '-' $\langle last \rangle$	no $\langle name \rangle.range$

Here  $\langle first \ expr \rangle$ ,  $\langle length \ expr \rangle$  and  $\langle end \ expr \rangle$  both denote algebraic expressions possibly involving named overlay specifications and counters. As integers, they respectively evaluate to  $\langle first \rangle$ ,  $\langle length \rangle$  and  $\langle last \rangle$ .

For example both  $?(\mathbf{A.next})$ ,  $?(\mathbf{A.last+1})$ ,  $?(\mathbf{A.1+A.length})$  give the same result as soon as the slide range named 'A' has been defined with a length.

1  $\langle *package \rangle$

## 6 Implementation

Identify the internal prefix ( $\text{\LaTeX3 DocStrip convention}$ ).

2  $\langle @@=beanover \rangle$

### 6.1 Package declarations

```

3 \NeedsTeXFormat{LaTeX2e}[2020/01/01]
4 \ProvidesExplPackage
5   {beanover}
6   {2022/10/28}
7   {1.0}
8   {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  $\text{\TeX}$  group, which ensures no name collision with the caller stack. In that case, variables need not follow exactly the  $\text{\LaTeX3}$  naming convention: we do not specialize with the module name. On execution, next group initialization instructions declare the variables as side effect.

```

9 \group_begin:
10 \tl_clear_new:N \l_a_tl
11 \tl_clear_new:N \l_b_tl
12 \tl_clear_new:N \l_c_tl
13 \tl_clear_new:N \l_ans_tl
14 \seq_clear_new:N \l_ans_seq
15 \seq_clear_new:N \l_match_seq
16 \seq_clear_new:N \l_token_seq
17 \int_zero_new:N \l_split_int
18 \seq_clear_new:N \l_split_seq
```

```

19 \int_zero_new:N \l_depth_int
20 \tl_clear_new:N \l_name_tl
21 \tl_clear_new:N \l_path_tl
22 \tl_clear_new:N \l_group_tl
23 \tl_clear_new:N \l_query_tl
24 \seq_clear_new:N \l_query_seq
25 \bool_set_false:N \l_no_counter_bool
26 \bool_set_false:N \l_no_range_bool
27 \group_end:

```

## 6.3 Overlay specification

### 6.3.1 In slide range definitions

`\g__beanover_prop`  $\langle key \rangle - \langle value \rangle$  property list to store the named slide lists. The basic keys are, assuming  $\langle name \rangle$  is a slide list identifier,

$\langle name \rangle / A$  for the first index

$\langle name \rangle / L$  for the length when provided

$\langle name \rangle / Z$  for the last index when provided

$\langle name \rangle / C$  for the counter value, when used

$\langle name \rangle / CO$  for initial value of the counter (when reset)

Other keys are eventually used to cache results when some attributes are defined from other slide ranges. They are characterized by a ‘//’.

$\langle name \rangle // A$  for the cached static value of the first index

$\langle name \rangle // Z$  for the cached static value of the last index

$\langle name \rangle // L$  for the cached static value of the length

$\langle name \rangle // N$  for the cached static value of the next index

The implementation is private, in particular, keys may change in future versions.

```

28 \prop_new:N \g__beanover_prop

```

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

---

<code>\__beanover_gput:nn</code> <code>\__beanover_gput:nV</code> <code>\__beanover_item:n</code> <code>\__beanover_get:nN</code> <code>\__beanover_gremove:n</code> <code>\__beanover_gclear:n</code> <code>\__beanover_gclear:</code>	<code>\__beanover_gput:nn {&lt;key&gt;} {&lt;value&gt;}</code> <code>\__beanover_item:n {&lt;key&gt;}</code> <code>\__beanover_get:n {&lt;key&gt;} &lt;tl variable&gt;</code> <code>\__beanover_gremove:n {&lt;key&gt;}</code> <code>\__beanover_gclear:n {&lt;key&gt;}</code> <code>\__beanover_gclear:</code>
---	--

---

Convenient shortcuts to manage the storage, it makes the code more concise and readable.

```

29 \cs_new:Npn \__beanover_gput:nn {
30   \prop_gput:Nnn \g__beanover_prop
31 }
32 \cs_new:Npn \__beanover_item:n {
33   \prop_item:Nn \g__beanover_prop
34 }
35 \cs_new:Npn \__beanover_get:nN {
36   \prop_get:NnN \g__beanover_prop
37 }
38 \cs_new:Npn \__beanover_gremove:n {
39   \prop_gremove:Nn \g__beanover_prop
40 }
41 \cs_new:Npn \__beanover_gclear:n #1 {
42   \clist_map_inline:nn { A, L, Z, C, CO, /A, /L, /Z, /N } {
43     \__beanover_gremove:n { #1 / ##1 }
44   }
45 }
46 \cs_new:Npn \__beanover_gclear: {
47   \prop_gclear:N \g__beanover_prop
48 }
49 \cs_generate_variant:Nn \__beanover_gput:nn { nV }

```

---

<code>\__beanover_if_in_p:n *</code> <code>\__beanover_if_in_p:V *</code> <code>\__beanover_if_in:nTF *</code> <code>\__beanover_if_in:VTF *</code>	<code>\__beanover_if_in_p:n {&lt;key&gt;}</code> <code>\__beanover_if_in:nTF {&lt;key&gt;} {&lt;true code&gt;} {&lt;false code&gt;}</code>
--	---

---

Convenient shortcuts to test for the existence of some key, it makes the code more concise and readable.

```

50 \prg_new_conditional:Npnn \__beanover_if_in:n #1 { p, T, F, TF } {
51   \prop_if_in:NnTF \g__beanover_prop { #1 } {
52     \prg_return_true:
53   } {
54     \prg_return_false:
55   }
56 }
57 \prg_generate_conditional_variant:Nnn \__beanover_if_in:n {V} { p, T, F, TF }

```

---

```
\__beanover_get:nNTF {<key>} {<tl variable>} {<true code>} {<false code>}
```

---

Convenient shortcuts to retrieve the value with branching, it makes the code more concise and readable. Execute *<true code>* when the item is found, *<false code>* otherwise. In the latter case, the content of the *<tl variable>* is undefined.

```
58 \prg_new_conditional:Npnn \__beanover_get:nN #1 #2 { T, F, TF } {
59   \prop_get:NnNTF \g__beanover_prop { #1 } #2 {
60     \prg_return_true:
61   } {
62     \prg_return_false:
63   }
64 }
```

Utility message.

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

### 6.3.2 Regular expressions

`\c__beanover_name_regex` The name of a slide range consists of a non void list of alphanumerical characters and underscore, but with no leading digit.

```
66 \regex_const:Nn \c__beanover_name_regex {
67   [[[:alpha:]]_][[:alnum:]]_*
68 }
```

(End definition for `\c__beanover_name_regex`.)

`\c__beanover_path_regex` A sequence of *<positive integer>* items representing a path.

```
69 \regex_const:Nn \c__beanover_path_regex {
70   (?: \. \d+ )_*
71 }
```

(End definition for `\c__beanover_path_regex`.)

`\c__beanover_key_regex` A key is the name of a slide range possibly followed by positive integer attributes using a dot syntax. The ‘A\_key\_Z’ variant matches the whole string.

```
72 \regex_const:Nn \c__beanover_key_regex {
73   \ur{c__beanover_name_regex} \ur{c__beanover_path_regex}
74 }
75 \regex_const:Nn \c__beanover_A_key_Z_regex {
76   \A \ur{c__beanover_key_regex} \Z
77 }
```

(End definition for `\c__beanover_key_regex` and `\c__beanover_A_key_Z_regex`.)

`\c__beanover_dotted_regex` A specifier is the name of a slide range possibly followed by attributes using a dot syntax. This is a poor man version to save computations, a dedicated parser would help in error management.

```
78 \regex_const:Nn \c__beanover_dotted_regex {
79   \A \ur{c__beanover_name_regex} (?: \. [^\.]+ )_* \Z
80 }
```

(End definition for `\c__beanover_dotted_regex`.)



`\c__beanover_colons_regex` For ranges defined by a colon syntax.

```
81 \regex_const:Nn \c__beanover_colons_regex { :(:+)? }
```

*(End definition for \c\_\_beanover\_colons\_regex.)*

`\c__beanover_int_regex` A decimal integer with an eventual leading sign next to the first digit.

```
82 \regex_const:Nn \c__beanover_int_regex {  
83   (?:[-+])? \d+  
84 }
```

*(End definition for \c\_\_beanover\_int\_regex.)*

`\c__beanover_list_regex` A comma separated list between square brackets.

```
85 \regex_const:Nn \c__beanover_list_regex {  
86   \A \[ \s*
```

Capture groups:

- 2: the content between the brackets, outer spaces trimmed out

```
87   ( [^\]]*? )  
88   \s* \] \Z  
89 }
```

*(End definition for \c\_\_beanover\_list\_regex.)*

`\c__beanover_split_regex` Used to parse slide list overlay specifications in queries. Next are the 10 capture groups. Group numbers are 1 based because the regex is used in splitting contexts where only capture groups are considered and not the whole match.

```
90 \regex_const:Nn \c__beanover_split_regex {  
91   \s* ( ? :
```

We start with ‘++’ instrussions <sup>1</sup>.

- 1:  $\langle name \rangle$  of a slide range

```
92   \+\+ ( \ur{c__beanover_name_regex} )
```

- 2: optionally followed by an integer path

```
93   ( \ur{c__beanover_path_regex} ) (?: \. n )?
```

We continue with other expressions

- 3:  $\langle name \rangle$  of a slide range

```
94   | ( \ur{c__beanover_name_regex} )
```

- 4: optionally followed by an integer path

```
95   ( \ur{c__beanover_path_regex} )
```

Next comes another branching

```
96   (?:
```

---

<sup>1</sup>At the same time an instruction and an expression... this is a synonym of exprection

- 5: the  $\langle length \rangle$  attribute

97        \. 1(e)ngth

- 6: the  $\langle last \rangle$  attribute

98        | \. 1(a)st

- 7: the  $\langle next \rangle$  attribute

99        | \. ne(x)t

- 8: the  $\langle range \rangle$  attribute

100       | \. (r)ange

- 9: the  $\langle n \rangle$  attribute

101       | \. (n)

• 10: the poor man integer expression after ‘+=’. When it contains no parenthesis, it is an algebraic expression involving integers and  $\langle key \rangle$ ’s. Otherwise it starts with a parenthesis and ends with the first parenthesis followed by a white space or the end of the text. This tricky definition allows quite any algebraic expression involving parenthesis. The problems may arise when dealing with nested expressions.

```
102      (? : \s* \+= \s*
103      ( (? : \ur{c__beanover_int_regex} | \ur{c__beanover_key_regex} )
104      (? : [\+\-*/] (? : \d+ | \ur{c__beanover_key_regex} ) ) *
105      | \ ( . * ? \ ) (? : \Z | \s+ )
106      )
107      ) ?

108      ) ?

109      ) \s*
110      }
```

(End definition for `\c__beanover_split_regex`.)

### 6.3.3 Defining named slide ranges

---

`\__beanover_error:n`

---

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

```
111 \cs_new:Npn \__beanover_error:n #1 {
112   \msg_fatal:nnn { beanover } { :n } { Missing~value~for~#1 }
113 }
```

---

`\__beanover_parse:nn`

---

`\__beanover_parse:nn { $\langle key \rangle$ } { $\langle definition \rangle$ }`

Auxiliary function called within a group.  $\langle name \rangle$  is the slide key, including eventually a dotted integer path,  $\langle definition \rangle$  is the corresponding definition.

`\l_match_seq` Local storage for the match result.

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

---

```

\__beanover_range:nnnn
\__beanover_range:nVVV
\__beanover_range:nnnnn
\__beanover_range:nVVVV

```

---

```

\__beanover_range:nnnn {<key>} {<first>} {<length>} {<last>}
\__beanover_range:nnnnn {<name>} {<path>} {<first>} {<length>} {<last>}

```

Auxiliary function called within a group. Setup the model to define a range.

```

114 \cs_new:Npn \__beanover_range:nnnn #1 #2 #3 #4 {
115   \__beanover_gclear:n { #1 }
116   \tl_if_empty:nTF { #2 } {
117     \tl_if_empty:nTF { #3 } {
118       \tl_if_empty:nTF { #4 } {
119         \msg_error:nnn { beanover } { :n } { Not-a-range::~~#1 }
120       } {
121         \__beanover_gput:nn { #1/Z } { #4 }
122       }
123     } {
124       \__beanover_gput:nn { #1/L } { #3 }
125       \tl_if_empty:nF { #4 } {
126         \__beanover_gput:nn { #1/Z } { #4 }
127         \__beanover_gput:nn { #1/A } { #1.last - (#1.length) + 1 }
128       }
129     }
130   } {
131     \__beanover_gput:nn { #1/A } { #2 }
132     \tl_if_empty:nTF { #3 } {
133       \tl_if_empty:nF { #4 } {
134         \__beanover_gput:nn { #1/Z } { #4 }
135         \__beanover_gput:nn { #1/L } { #1.last - (#1.first) + 1 }
136       }
137     } {
138       \__beanover_gput:nn { #1/L } { #3 }
139       \__beanover_gput:nn { #1/Z } { #1.first + #1.length - 1 }
140     }
141   }
142 }
143 \cs_generate_variant:Nn \__beanover_range:nnnn { nVVV }

144 \cs_generate_variant:Nn \tl_if_empty:nTF { xTF }
145 \cs_new:Npn \__beanover_do_parse:nn #1 #2 {

```

This is not a list.

```

146   \tl_clear:N \l_a_tl
147   \tl_clear:N \l_b_tl
148   \tl_clear:N \l_c_tl
149   \regex_split:NnN \c__beanover_colons_regex { #2 } \l_split_seq
150   \seq_pop_left:NNT \l_split_seq \l_a_tl {

```

\l\_a\_tl may contain the *<start>*.

```

151   \seq_pop_left:NNT \l_split_seq \l_b_tl {
152     \tl_if_empty:nTF \l_b_tl {

```

This is a one colon range.

```

153     \seq_pop_left:NN \l_split_seq \l_b_tl

```

\l\_b\_tl may contain the *<length>*.

```

154     \seq_pop_left:NNT \l_split_seq \l_c_tl {
155       \tl_if_empty:nTF \l_c_tl {

```

A :: was expected:

```

156 \msg_error:nnn { beanover } { :n } { Invalid-range-expression(1):~#2 }
157     } {
158         \int_compare:nNt { \tl_count:N \l_c_tl } > { 1 } {
159 \msg_error:nnn { beanover } { :n } { Invalid-range-expression(2):~#2 }
160     }
161     \seq_pop_left:NN \l_split_seq \l_c_tl

```

\l\_c\_tl may contain the  $\langle end \rangle$ .

```

162         \seq_if_empty:NF \l_split_seq {
163 \msg_error:nnn { beanover } { :n } { Invalid-range-expression(3):~#2 }
164     }
165     }
166 }
167 } {

```

This is a two colon range.

```

168         \int_compare:nNt { \tl_count:N \l_b_tl } > { 1 } {
169 \msg_error:nnn { beanover } { :n } { Invalid-range-expression(4):~#2 }
170     }
171     \seq_pop_left:NN \l_split_seq \l_c_tl

```

\l\_c\_tl contains the  $\langle end \rangle$ .

```

172         \seq_pop_left:NNTF \l_split_seq \l_b_tl {
173         \tl_if_empty:NTF \l_b_tl {
174         \seq_pop_left:NN \l_split_seq \l_b_tl

```

\l\_b\_tl may contain the  $\langle length \rangle$ .

```

175         \seq_if_empty:NF \l_split_seq {
176 \msg_error:nnn { beanover } { :n } { Invalid-range-expression(5):~#2 }
177     }
178     } {
179 \msg_error:nnn { beanover } { :n } { Invalid-range-expression(6):~#2 }
180     }
181     } {
182         \tl_clear:N \l_b_tl
183     }
184 }
185 }
186 }

```

Providing both the  $\langle start \rangle$ ,  $\langle length \rangle$  and  $\langle end \rangle$  of a range is not allowed, even if they happen to be consistent.

```

187 \bool_if:nF {
188     \tl_if_empty_p:N \l_a_tl
189     || \tl_if_empty_p:N \l_b_tl
190     || \tl_if_empty_p:N \l_c_tl
191 } {
192 \msg_error:nnn { beanover } { :n } { Invalid-range-expression(7):~#2 }
193 }
194 \__beanover_range:nVVV { #1 } \l_a_tl \l_b_tl \l_c_tl
195 }

196 \cs_new:Npn \__beanover_parse:nn #1 #2 {
197     \group_begin:
198     \regex_match:NnTF \c__beanover_A_key_Z_regex { #1 } {

```

We got a valid key.

```

199 \regex_extract_once:NnNTF \c__beanover_list_regex { #2 } \l_match_seq {
This is a comma separated list, extract each item and go recursive.
200 \exp_args:NNx
201 \seq_set_from_clist:Nn \l_match_seq {
202 \seq_item:Nn \l_match_seq { 2 }
203 }
204 \seq_map_indexed_inline:Nn \l_match_seq {
205 \__beanover_do_parse:nn { #1.##1 } { ##2 }
206 }
207 } {
208 \__beanover_do_parse:nn { #1 } { #2 }
209 }
210 } {
211 \msg_error:nnn { beanover } { :n } { Invalid-key:~#1 }
212 }
213 \group_end:
214 }

```

---

**\Beanover** \Beanover {*<key--value list>*}

---

The keys are the slide range specifiers. 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.

```

215 \NewDocumentCommand \Beanover { m } {
216 \keyval_parse:NNn \__beanover_error:n \__beanover_parse:nn { #1 }
217 \ignorespaces
218 }

```

If we use this command in the frame body, it will be executed for each different frame. If we use the frame option `beanover` instead, the command is executed only once, at the cost of a more verbose code.

```

219 \define@key{beamerframe}{beanover}{\Beanover{#1}}

```

### 6.3.4 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.

```

220 \cs_set_eq:NN \__beanover_beamer@masterdecode \beamer@masterdecode
221 \cs_set:Npn \beamer@masterdecode #1 {
222 \group_begin:
223 \tl_clear:N \l_ans_tl
224 \__beanover_scan:nNN { #1 } \__beanover_eval:nN \l_ans_tl

```

```

225 \exp_args:NNV
226 \group_end:
227 \__beanover_beamer@masterdecode \l_ans_tl
228 }

```

---

**\\_\_beanover\_scan:nNN**

---

**\\_\_beanover\_scan:nNN** {*⟨named overlay expression⟩*} *⟨eval⟩* *⟨tl variable⟩*

Scan the *⟨named overlay expression⟩* argument and feed the *⟨tl variable⟩* replacing *?(...)* instructions by their static counterpart with help from the *⟨eval⟩* function, which is **\\_\_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\_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.

*(End definition for \l\_ask\_bool. This variable is documented on page ??.)*

**\l\_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.

*(End definition for \l\_query\_bool. This variable is documented on page ??.)*

**\l\_token\_tl** Storage for just one token.

*(End definition for \l\_token\_tl. This variable is documented on page ??.)*

```

229 \cs_new:Npn \__beanover_scan:nNN #1 #2 #3 {
230   \group_begin:
231   \tl_clear:N \l_ans_tl
232   \int_zero:N \l_depth_int
233   \seq_clear:N \l_token_seq

```

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

```

234   \regex_split:nnN {} { #1 } \l_token_seq

```

Run the top level loop to scan for a ‘?’:

```

235   \bool_set_true:N \l_ask_bool
236   \bool_while_do:Nn \l_ask_bool {
237     \seq_pop_left:NN \l_token_seq \l_token_tl
238     \quark_if_no_value:NTF \l_token_tl {

```

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

```
239     \bool_set_false:N \l_ask_bool
240   } {
```

\l\_token\_tl contains a ‘normal’ token.

```
241     \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.

```
242         \bool_set_true:N \l_query_bool
243         \bool_while_do:Nn \l_query_bool {
```

Get next token.

```
244             \seq_pop_left:NN \l_token_seq \l_token_tl
245             \quark_if_no_value:Ntf \l_token_tl {
```

No opening parenthesis found, raise.

```
246                 \msg_fatal:nxx { beanover } { :n } {Missing~'('%---)
247                 ~after~a~?:~#1}
248             } {
249                 \tl_if_eq:NnT \l_token_tl { ( % )
250             } {
```

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

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

Record the forthcoming content in the \l\_query\_tl variable, up to the next balancing ‘)’.

```
252                 \tl_clear:N \l_query_tl
253                 \bool_while_do:Nn \l_query_bool {
```

Get next token.

```
254                     \seq_pop_left:NN \l_token_seq \l_token_tl
255                     \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.

```
256             \msg_error:nxx { beanover } { :n } {Missing~%((---
257             `)'':~#1 }
258             \int_do_while:nNnn \l_depth_int > 1 {
259                 \int_decr:N \l_depth_int
260                 \tl_put_right:Nn \l_query_tl {%(---
261             )}
262             }
263             \int_zero:N \l_depth_int
264             \bool_set_false:N \l_query_bool
265             \bool_set_false:N \l_ask_bool
266         } {
267             \tl_if_eq:NnTF \l_token_tl { ( % ---)
268         } {
```

We found a ‘(’, increment the depth and append the token to \l\_query\_tl.

```
269                 \int_incr:N \l_depth_int
270                 \tl_put_right:NV \l_query_tl \l_token_tl
271             } {
```

This is not a ‘(’.

```

272             \tl_if_eq:NnTF \l_token_tl { %(
273             )
274         } {

```

We found a ‘)’, decrement the depth.

```

275             \int_decr:N \l_depth_int
276             \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.

```

277     \exp_args:NV #2 \l_query_tl \l_ans_tl
278     \bool_set_false:N \l_query_bool
279     } {

```

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

```

280             \tl_put_right:NV \l_query_tl \l_token_tl
281         }

```

Above ends the code for a positive depth.

```

282     } {

```

The scanned token is not a ‘(’ nor a ‘)’, we append it as is to \l\_query\_tl.

```

283             \tl_put_right:NV \l_query_tl \l_token_tl
284         }
285     }
286 }

```

Above ends the code for Not a ‘(’

```

287     }
288 }

```

Above ends the code for: Found the ‘(’ after the ‘?’

```

289     }

```

Above ends the code for not a no value quark.

```

290     }

```

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.

```

291     \quark_if_no_value:NT \l_token_tl {
292         \bool_set_false:N \l_query_bool
293         \bool_set_false:N \l_ask_bool
294     }
295 } {

```

This is not a ‘?’, append the token to right of \l\_ans\_tl and continue.

```

296     \tl_put_right:NV \l_ans_tl \l_token_tl
297 }

```

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

```

298     }
299 }

```



Above ends the outer bool while loop to find ‘?’ characters. We can append our result to  $\langle tl\ variable \rangle$

```

300 \exp_args:NNNV
301 \group_end:
302 \tl_put_right:Nn #3 \l_ans_tl
303 }

```

Each new frame has its own set of slide ranges, we clear the property list on entering a new frame environment. Frame environments nested into other frame environments are not supported.

```

304 \AddToHook
305 { env/beamer@framepauses/before }
306 { \prop_gclear:N \g__beanover_prop }

```

### 6.3.5 Evaluation bricks

---

```

\__beanover_if_first_p:nN * \__beanover_if_first:nTF {<name>} <tl variable> {<true code>} {<false
\__beanover_if_first:nTF * code>}

```

---

Append the first of the  $\langle name \rangle$  slide range to the  $\langle tl\ variable \rangle$ . Cache the result. Execute  $\langle true\ code \rangle$  when there is a  $\langle first \rangle$ ,  $\langle false\ code \rangle$  otherwise.

```

307 \prg_new_conditional:Npnn \__beanover_if_first:nN #1 #2 { p, T, F, TF } {
308 \__beanover_if_in:nTF { #1//A } {
309 \tl_put_right:Nx #2 { \__beanover_item:n { #1//A } }
310 \prg_return_true:
311 } {
312 \group_begin:
313 \tl_clear:N \l_ans_tl
314 \__beanover_if_in:nTF { #1/A } {
315 \__beanover_eval:xN {
316 \__beanover_item:n { #1/A }
317 } \l_ans_tl
318 } {
319 \bool_if:nTF {
320 \__beanover_if_in_p:n { #1/L } && \__beanover_if_in_p:n { #1/Z }
321 } {
322 \__beanover_eval:xN {
323 \__beanover_item:n { #1/Z } - ( \__beanover_item:n { #1/L } - 1 )
324 } \l_ans_tl
325 } {
326 \__beanover_if_in:nT { #1/C } {
327 \bool_set_true:N \l_no_counter_bool
328 \__beanover_eval:xN {
329 \__beanover_item:n { #1/C }
330 } \l_ans_tl
331 }
332 }
333 }
334 \tl_if_empty:NTF \l_ans_tl {
335 \group_end:
336 \prg_return_false:
337 } {
338 \__beanover_gput:nV { #1//A } \l_ans_tl

```

```

339     \exp_args:NNNV
340     \group_end:
341     \tl_put_right:Nn #2 \l_ans_tl
342     \prg_return_true:
343   }
344 }
345 }

```

---

```

\__beanover_first:nN
\__beanover_first:VN

```

---

```
\__beanover_first:nN {\langle name \rangle} \langle tl variable \rangle
```

Append the start of the  $\langle name \rangle$  slide range to the  $\langle tl variable \rangle$ . Cache the result.

```

346 \cs_new:Npn \__beanover_first:nN #1 #2 {
347   \__beanover_if_first:nNF { #1 } #2 {
348     \msg_error:nnn { beanover } { :n } { Range~with~no~first:~#1 }
349   }
350 }
351 \cs_generate_variant:Nn \__beanover_first:nN { VN }

```

FAILURE ‘X.last-(X.length)+1’!=‘C-(A-1)’

Test \\_\_beanover\_first:nN 3

FAILURE ‘X.last-(X.length)+1’!=‘C-(A-1)’

Test \\_\_beanover\_first:nN 4

FAILURE ‘X.last-(X.length)+1’!=‘C-(A-1)’

Test \\_\_beanover\_first:nN 5

FAILURE ‘X.last-(X.length)+1’!=‘C-(A-1)’

Test \\_\_beanover\_first:nN 6

---

```

\__beanover_if_length_p:nN * \__beanover_length_p:nN {\langle name \rangle} \langle tl variable \rangle
\__beanover_if_length:nNTF * \__beanover_if_length:nNTF {\langle name \rangle} \langle tl variable \rangle {\langle true code \rangle} {\langle false
code \rangle}

```

---

Append the length of the  $\langle name \rangle$  slide range to  $\langle tl variable \rangle$  Execute  $\langle true code \rangle$  when there is a  $\langle length \rangle$ ,  $\langle false code \rangle$  otherwise.

```

352 \prg_new_conditional:Npnn \__beanover_if_length:nN #1 #2 { p, T, F, TF } {
353   \__beanover_if_in:nTF { #1//L } {
354     \tl_put_right:Nx #2 { \__beanover_item:n { #1//L } }
355     \prg_return_true:
356   } {
357     \group_begin:
358     \tl_clear:N \l_ans_tl
359     \__beanover_if_in:nTF { #1/L } {
360       \__beanover_eval:xN {
361         \__beanover_item:n { #1/L }
362       } \l_ans_tl
363     } {
364       \bool_if:nT {
365         \__beanover_if_in_p:n { #1/A } && \__beanover_if_in_p:n { #1/Z }

```

```





366     } {
367         \__beanover_eval:xN {
368             \__beanover_item:n { #1/Z } - (\__beanover_item:n { #1/A } - 1)
369         } \l_ans_tl
370     }
371 }
372 \tl_if_empty:NTF \l_ans_tl {
373     \group_end:
374     \prg_return_false:
375 } {
376     \__beanover_gput:nV { #1//L } \l_ans_tl
377     \exp_args:NNNV
378     \group_end:
379     \tl_put_right:Nn #2 \l_ans_tl
380     \prg_return_true:
381 }
382 }
383 }

```

---

$\backslash\_beanover\_length:nN$ $\backslash\_beanover\_length:VN$	$\backslash\_beanover\_length:nN \{ \langle name \rangle \} \langle tl \ variable \rangle$ Append the length of the $\langle name \rangle$ slide range to $\langle tl \ variable \rangle$
--	--





---

-  **FAILURE** ‘X.last-(X.first)+1’!=‘C-(A-1)’
-  **Test**  $\backslash\_beanover\_length:nN \ 2$
-  **FAILURE** ‘X.last-(X.first)+1’!=‘C-(A-1)’
-  **Test**  $\backslash\_beanover\_length:nN \ 2$

```

384 \cs_new:Npn \__beanover_length:nN #1 #2 {
385     \__beanover_if_length:nNF { #1 } #2 {
386         \msg_error:nnn { beanover } { :n } { Range~with~no~length::~#1 }
387     }
388 }
389 \cs_generate_variant:Nn \__beanover_length:nN { VN }

```

-  **FAILURE** ‘X.last-(X.first)+1’!=‘B-(A-1)’
-  **Test**  $\backslash\_beanover\_length:nN \ 3$
-  **FAILURE** ‘X.last-(X.first)+1’!=‘B-(A-1)’
-  **Test**  $\backslash\_beanover\_length:nN \ 4$

---

$\backslash\_beanover\_if\_last\_p:nN \star$ $\backslash\_beanover\_if\_last:nN \underline{TF} \star$	$\backslash\_beanover\_if\_last\_p:nN \{ \langle name \rangle \} \langle tl \ variable \rangle$ $\backslash\_beanover\_if\_last:nN \underline{TF} \{ \langle name \rangle \} \langle tl \ variable \rangle \{ \langle true \ code \rangle \} \{ \langle false \ code \rangle \}$
--	---

---

```

390 \prg_new_conditional:Npnn \__beanover_if_last:nN #1 #2 { p, T, F, TF } {
391     \__beanover_if_in:nTF { #1//Z } {
392         \tl_put_right:Nx #2 { \__beanover_item:n { #1//Z } }
393         \prg_return_true:

```

```

394 } {
395   \group_begin:
396   \tl_clear:N \l_ans_tl
397   \__beanover_if_in:nTF { #1/Z } {
398     \__beanover_eval:xN {
399       \__beanover_item:n { #1/Z }
400     } \l_ans_tl
401   } {
402     \__beanover_get:nNT { #1/A } \l_a_tl {
403       \__beanover_get:nNT { #1/L } \l_b_tl {
404         \__beanover_eval:xN {
405           \l_a_tl + \l_b_tl - 1
406         } \l_ans_tl
407       }
408     }
409   }
410   \tl_if_empty:NTF \l_ans_tl {
411     \group_end:
412     \prg_return_false:
413   } {
414     \__beanover_gput:nV { #1//Z } \l_ans_tl
415     \exp_args:NNNV
416     \group_end:
417     \tl_put_right:Nn #2 \l_ans_tl
418     \prg_return_true:
419   }
420 }
421 }

```

---

`\__beanover_last:nN`  
`\__beanover_last:VN`









---

`\__beanover_last:nN {<name>} <tl variable>`  
Append the last index of the `<name>` slide range to `<tl variable>`

```

422 \cs_new:Npn \__beanover_last:nN #1 #2 {
423   \__beanover_if_last:nNF { #1 } #2 {
424     \msg_error:nnn { beanover } { :n } { Range~with~no~last:~#1 }
425   }
426 }
427 \cs_generate_variant:Nn \__beanover_last:nN { VN }

```

-  **FAILURE** ‘X.first+X.length-1’!=‘A+B-1’
-  **Test** `\__beanover_last:nN 5-a`
-  **FAILURE** ‘X.first+X.length-1’!=‘A+B-1’
-  **Test** `\__beanover_last:nN 5-c`
-  **FAILURE** ‘X.first+X.length-1’!=‘A+B-1’
-  **Test** `\__beanover_last:nN 5-a`
-  **FAILURE** ‘X.first+X.length-1’!=‘A+B-1’
-  **Test** `\__beanover_last:nN 5-c`

- ❌ FAILURE 'X.first+X.length-1'!='A+B-1'
- ❌ Test \\_beanover\\_last:nN 6-a
- ❌ FAILURE 'X.first+X.length-1'!='A+B-1'
- ❌ Test \\_beanover\\_last:nN 6-c
- ❌ FAILURE 'X.first+X.length-1'!='A+B-1'
- ❌ Test \\_beanover\\_last:nN 6-a
- ❌ FAILURE 'X.first+X.length-1'!='A+B-1'
- ❌ Test \\_beanover\\_last:nN 6-c

---

\\_beanover\\_if\\_next\\_p:nN ★  
 \\_beanover\\_if\\_next:nN $\overline{TF}$  ★

---

\\_beanover\\_if\\_next\\_p:nN {<name>} <tl variable>  
 \\_beanover\\_if\\_next:nNTF {<name>} <tl variable> {<true code>} {<false code>}

Append the index after the <name> slide range to the <tl variable>. Execute <true code> when there is a <next> index, <false code> otherwise.

```

428 \prg_new_conditional:Npnn \_beanover\_if\_next:nN #1 #2 { p, T, F, TF } {
429   \_beanover\_if\_in:nTF { #1//N } {
430     \tl\_put\_right:Nx #2 { \_beanover\_item:n { #1//N } }
431     \prg\_return\_true:
432   } {
433     \group\_begin:
434     \_beanover\_get:nNTF { #1/Z } \l\_ans\_tl {
435       \tl\_put\_right:Nn \l\_ans\_tl { +1 }
436     } {
437       \_beanover\_get:nNT { #1/A } \l\_a\_tl {
438         \_beanover\_get:nNT { #1/L } \l\_b\_tl {
439           \_beanover\_eval:xN {
440             \l\_a\_tl + \l\_b\_tl
441           } \l\_ans\_tl
442         }
443       }
444     }
445     \tl\_if\_empty:NNTF \l\_ans\_tl {
446       \group\_end:
447       \prg\_return\_false:
448     } {
449       \_beanover\_gput:nV { #1//N } \l\_ans\_tl
450       \exp\_args:NNNV
451       \group\_end:
452       \tl\_put\_right:Nn #2 \l\_ans\_tl
453       \prg\_return\_true:
454     }
455   }
456 }
```

---

\\_beanover\\_next:nN  
 \\_beanover\\_next:VN

---

\\_beanover\\_next:nN {<name>} <tl variable>

Append the index after the <name> slide range to the <tl variable>.

```

457 \cs_new:Npn \__beanovery_next:nN #1 #2 {
458   \__beanovery_if_next:nNF { #1 } #2 {
459     \msg_error:nnn { beanovery } { :n } { Range~with~no~next:~#1 }
460   }
461 }
462 \cs_generate_variant:Nn \__beanovery_next:nN { VN }

```

- ❌ FAILURE ‘X.first+X.length-1+1’!=‘A+B’
- ❌ Test \\_\_beanovery\_next:nN 3-a
- ❌ FAILURE ‘X.first+X.length-1+1’!=‘A+B’
- ❌ Test \\_\_beanovery\_next:nN 3-c
- ❌ FAILURE ‘X.first+X.length-1+1’!=‘A+B’
- ❌ Test \\_\_beanovery\_next:nN 3-a
- ❌ FAILURE ‘X.first+X.length-1+1’!=‘A+B’
- ❌ Test \\_\_beanovery\_next:nN 3-c

---

```

\__beanovery_free_counter:nN \__beanovery_free_counter:nN {<name>} <tl variable>
\__beanovery_free_counter:VN

```

---

Append the value of the counter associated to the  $\{\langle name \rangle\}$  slide range to the right of  $\langle tl variable \rangle$ . There is no branching variant because, we always return some value, ‘1’ by default.

```

463 \cs_new:Npn \__beanovery_free_counter:nN #1 #2 {
464   \group_begin:
465   \tl_clear:N \l_ans_tl
466   \__beanovery_get:nNF { #1/C } \l_ans_tl {
467     \__beanovery_if_first:nNF { #1 } \l_ans_tl {
468       \__beanovery_if_last:nNF { #1 } \l_ans_tl {
469         \tl_set:Nn \l_ans_tl { 1 }
470       }
471     }
472   }
473   \__beanovery_gput:nV { #1/C } \l_ans_tl
474   \exp_args:NNNV
475   \group_end:
476   \tl_put_right:Nn #2 \l_ans_tl
477 }
478 \cs_generate_variant:Nn \__beanovery_free_counter:nN { VN }

```

---

```

\__beanovery_counter:nN \__beanovery_counter:nN {<name>} <tl variable>
\__beanovery_counter:VN

```

---

Append the value of the counter associated to the  $\{\langle name \rangle\}$  slide range to the right of  $\langle tl variable \rangle$ . The value always lays in between the range, whenever possible.

```

479 \cs_new:Npn \__beanovery_counter:nN #1 #2 {
480   \group_begin:
481   \__beanovery_free_counter:nN { #1 } \l_ans_tl

```

If there is a  $\langle first \rangle$ , use it to bound the result from below.

```

482 \tl_clear:N \l_a_tl
483 \__beanover_if_first:nNT { #1 } \l_a_tl {
484   \fp_compare:nNt { \l_ans_tl } < { \l_a_tl } {
485     \tl_set:NV \l_ans_tl \l_a_tl
486   }
487 }

```

If there is a  $\langle last \rangle$ , use it to bound the result from above.

```

488 \tl_clear:N \l_a_tl
489 \__beanover_if_last:nNT { #1 } \l_a_tl {
490   \fp_compare:nNt { \l_ans_tl } > { \l_a_tl } {
491     \tl_set:NV \l_ans_tl \l_a_tl
492   }
493 }
494 \exp_args:NNNx
495 \group_end:
496 \tl_set:Nn #2 { \fp_eval:n { round(\l_ans_tl) } }
497 }
498 \cs_generate_variant:Nn \__beanover_counter:nN { VN }

```

---

```

\__beanover_index:nnN
\__beanover_index:VVN

```

---

$\__beanover\_index:nnN$   $\{\langle name \rangle\}$   $\{\langle integer\ path \rangle\}$   $\langle tl\ variable \rangle$

Append the value of the counter associated to the  $\{\langle name \rangle\}$  slide range to the right of  $\langle tl\ variable \rangle$ . The value always lays in between the range, whenever possible.

```

499 \cs_new:Npn \__beanover_index:nnN #1 #2 #3 {
500   \group_begin:
501   \tl_set:Nn \l_name_tl { #1 }
502   \regex_split:nnNTF { \. } { #2 } \l_split_seq {
503     \seq_pop_left:NN \l_split_seq \l_a_tl
504     \seq_pop_right:NN \l_split_seq \l_b_tl
505     \seq_map_inline:Nn \l_split_seq {
506       \tl_set_eq:NN \l_b_tl \l_name_tl
507       \tl_put_right:Nn \l_b_tl { . ##1 }
508       \exp_args:Nx
509       \__beanover_get:nN { \l_b_tl / A } \l_c_tl
510       \quark_if_no_value:NTF \l_c_tl {
511         \tl_set_eq:NN \l_name_tl \l_b_tl
512       } {
513         \tl_set_eq:NN \l_name_tl \l_c_tl
514       }
515     }
516   } {
517     \msg_error:nnx { beanover } { :n } { Internal~error (#{1}/#{2}) }
518   }
519   \tl_clear:N \l_b_tl
520   \exp_args:Nx
521   \__beanover_get:nN { \l_name_tl.\l_a_tl / A } \l_b_tl
522   \quark_if_no_value:NTF \l_b_tl {
523     \exp_args:NV
524     \__beanover_first:nN \l_name_tl \l_ans_tl
525     \tl_put_right:Nx \l_ans_tl { + \l_a_tl - 1 }
526   } {
527     \tl_set_eq:NN \l_ans_tl \l_b_tl

```

```

528 }
529 \exp_args:NNNx
530 \group_end:
531 \tl_set:Nn #3 { \fp_eval:n { round(\l_ans_tl) } }
532 }

```

---

```

\__beanover_incr:nn
\__beanover_incr:nnN
\__beanover_incr:(VnN|VVN)

```

---

```

\__beanover_incr:nn {<name>} {<offset>}
\__beanover_incr:nnN {<name>} {<offset>} <tl variable>

```

Increment the free counter position accordingly. When requested, put the result in the *<tl variable>*. The result will lay within the declared range.

```

533 \cs_new:Npn \__beanover_incr:nn #1 #2 {
534   \group_begin:
535   \tl_clear:N \l_a_tl
536   \__beanover_free_counter:nN { #1 } \l_a_tl
537   \tl_clear:N \l_ans_tl
538   \__beanover_eval:xN { \l_a_tl + ( #2 ) } \l_ans_tl
539   \__beanover_gput:nV { #1/C } \l_ans_tl
540   \group_end:
541 }
542 \cs_new:Npn \__beanover_incr:nnN #1 #2 #3 {
543   \__beanover_incr:nn { #1 } { #2 }
544   \__beanover_counter:nN { #1 } #3
545 }
546 \cs_generate_variant:Nn \__beanover_incr:nnN { VnN }
547 \cs_generate_variant:Nn \__beanover_incr:nnN { VVN }

```

### 6.3.6 Evaluation

---

```

\__beanover_resolve:nnN
\__beanover_resolve:VVN

```

---

```

\__beanover_resolve:nnN {<name>} {<path>} <tl variable>

```

Resolve the *<name>* and *<path>* into a key that is put into the *<tl variable>*.

```

548 \cs_new:Npn \__beanover_resolve:nnN #1 #2 #3 {
549   \group_begin:
550   \tl_set:Nn \l_a_tl { #1 }
551   \regex_split:nnNT { \. } { #2 } \l_split_seq {
552     \seq_pop_left:NN \l_split_seq \l_b_tl
553     \seq_map_inline:Nn \l_split_seq {
554       \tl_set_eq:NN \l_b_tl \l_a_tl
555       \tl_put_right:Nn \l_b_tl { . ##1 }
556       \exp_args:Nx
557       \__beanover_get:nN { \l_b_tl / A } \l_c_tl
558       \quark_if_no_value:NTF \l_c_tl {
559         \tl_set_eq:NN \l_a_tl \l_b_tl
560       } {
561         \tl_set_eq:NN \l_a_tl \l_c_tl
562       }
563     }
564   }
565   \exp_args:NNNV
566   \group_end:

```



```

567 \tl_set:Nn #3 \l_a_tl
568 }
569 \cs_generate_variant:Nn \__beanover_resolve:nnN { VVN }

```

---

`\__beanover_append:nN`  
`\__beanover_append:VN`

---

`\__beanover_append:nN`  $\langle integer\ expression \rangle$   $\langle tl\ variable \rangle$

Evaluates the  $\langle integer\ expression \rangle$ , replacing all the named specifications by their static counterpart then put the result to the right of the  $\langle tl\ variable \rangle$ . Executed within a group. Heavily used by `\__beanover_eval_query:nN`, where  $\langle integer\ expression \rangle$  was enclosed in ‘ $?(...)$ ’. Local variables:

`\l_ans_tl` For the content of  $\langle tl\ variable \rangle$

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

`\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 caught 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.

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

`\l_path_tl` Storage for `\l_split_seq` items that represent paths.

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

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

(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 ??.)

```

570 \cs_new:Npn \__beanover_append:nN #1 #2 {
571   \group_begin:

```

Local variables:

```

572 \tl_clear:N \l_ans_tl
573 \int_zero:N \l_split_int
574 \seq_clear:N \l_split_seq
575 \tl_clear:N \l_name_tl
576 \tl_clear:N \l_path_tl
577 \tl_clear:N \l_group_tl
578 \tl_clear:N \l_a_tl

```

Implementation:

```

579 \regex_split:NnN \c__beanover_split_regex { #1 } \l_split_seq
580 \int_set:Nn \l_split_int { 1 }
581 \tl_set:Nx \l_ans_tl { \seq_item:Nn \l_split_seq { \l_split_int } }

```

---

```
\switch:nTF {\capture group number} {\black code} {\white code}
```

---

Helper function to locally set the `\l_group_tl` variable to the captured group *capture group number* and branch.

```
582 \cs_set:Npn \switch:nTF ##1 ##2 ##3 ##4 {
583   \tl_set:Nx ##2 {
584     \seq_item:Nn \l_split_seq { \l_split_int + ##1 }
585   }
586   \tl_if_empty:NTF ##2 { ##4 } { ##3 }
587 }
```

Main loop.

```
588 \int_while_do:nNnn { \l_split_int } < { \seq_count:N \l_split_seq } {
589   \switch:nTF 1 \l_name_tl {
```

- Case ++*name*<*integer path*>.n.

```
590   \switch:nTF 2 \l_path_tl {
591     \__beanover_resolve:VVN \l_name_tl \l_path_tl \l_name_tl
592   } { }
593   \__beanover_incr:VnN \l_name_tl 1 \l_ans_tl
594 } {
595   \switch:nTF 3 \l_name_tl {
```

- Cases *name*<*integer path*>....

```
596   \tl_set:Nn \l_b_tl {
597     \switch:nTF 4 \l_path_tl {
598       \__beanover_resolve:VVN \l_name_tl \l_path_tl \l_name_tl
599     } { }
600   }
601   \switch:nTF 5 \l_a_tl {
```

- Case ...length.

```
602   \l_b_tl
603   \__beanover_length:VN \l_name_tl \l_ans_tl
604 } {
605   \switch:nTF 6 \l_a_tl {
```

- Case ...last.

```
606   \l_b_tl
607   \__beanover_last:VN \l_name_tl \l_ans_tl
608 } {
609   \switch:nTF 7 \l_a_tl {
```

- Case ...next.

```
610   \l_b_tl
611   \__beanover_next:VN \l_name_tl \l_ans_tl
612 } {
613   \switch:nTF 8 \l_a_tl {
```

- Case ...range.

```

614         \l_b_tl
615         \__beanover_range:VN \l_name_tl \l_ans_tl
616     } {
617         \switch:nNTF 9 \l_a_tl {

```

- Case ...n.

```

618         \l_b_tl
619         \switch:nNTF { 10 } \l_a_tl {

```

- Case ...+=*<integer>*.

```

620         \__beanover_incr:VVN \l_name_tl \l_a_tl \l_ans_tl
621     } {
622         \__beanover_counter:VN \l_name_tl \l_ans_tl
623     }
624     } {
625         \switch:nNTF 4 \l_path_tl {
626             \exp_args:NVV
627             \__beanover_counter:nnN \l_name_tl \l_path_tl \l_ans_tl
628         } {
629             \exp_args:NV
630             \__beanover_counter:nnN \l_name_tl { .1 } \l_ans_tl
631         }
632     }
633 }
634 }
635 }
636 }
637 }
638 }
639 \int_add:Nn \l_split_int { 11 }
640 \tl_put_right:Nx \l_ans_tl { \seq_item:Nn \l_split_seq { \l_split_int } }
641 }
642 \exp_args:NNNx
643 \group_end:
644 \tl_put_right:Nn #2 { \fp_to_int:n { \l_ans_tl } }
645 }
646 \cs_generate_variant:Nn \__beanover_append:nN { VN }

```

<u>\__beanover_eval_query:nN</u>	<p><code>\__beanover_eval_query:Nn {&lt;overlay query&gt;} &lt;seq variable&gt;</code></p> <p>Evaluates the single <i>&lt;overlay query&gt;</i>, which is expected to contain no comma. Extract a range specification from the argument, replaces all the named overlay specifications by their static counterparts, make the computation then append the result to the right of the <i>&lt;seq variable&gt;</i>. Ranges are supported with the colon syntax. This is executed within a local group. Below are local variables and constants.</p> <p><code>\l_a_tl</code> Storage for the first index of a range.</p> <p><i>(End definition for \l_a_tl. This variable is documented on page ??.)</i></p> <p><code>\l_b_tl</code> Storage for the last index of a range, or its length.</p> <p><i>(End definition for \l_b_tl. This variable is documented on page ??.)</i></p> <p><code>\c__beanover_A_cln_Z_regex</code> Used to parse slide range overlay specifications. Next are the capture groups.</p> <p><i>(End definition for \c__beanover_A_cln_Z_regex.)</i></p> <pre> 647 \regex_const:Nn \c__beanover_A_cln_Z_regex { 648   \A \s* (?:         • 2: &lt;first&gt; 649       ( [^:]* ) \s* :         • 3: second optional colon 650       (:)? \s*         • 4: &lt;length&gt; 651       ( [^:]* )         • 5: standalone &lt;first&gt; 652         ( [^:]+ ) 653   ) \s* \Z 654 }</pre> <pre> 655 \cs_new:Npn \__beanover_eval_query:nN #1 #2 { 656   \regex_extract_once:NnNTF \c__beanover_A_cln_Z_regex { 657     #1 658   } \l_match_seq { 659     \tl_clear:N \l_ans_tl 660     \bool_set_false:N \l_no_counter_bool 661     \bool_set_true:N \l_no_range_bool</pre>
<u>\switch:nNTF</u>	<p><code>\switch:nNTF {&lt;capture group number&gt;} &lt;tl variable&gt; {&lt;black code&gt;} {&lt;white code&gt;}</code></p> <p>Helper function to locally set the <i>&lt;tl variable&gt;</i> to the captured group <i>&lt;capture group number&gt;</i> and branch depending on the emptiness of this variable.</p> <pre> 662   \cs_set:Npn \switch:nNTF ##1 ##2 ##3 ##4 { 663     \tl_set:Nx ##2 { 664       \seq_item:Nn \l_split_seq { ##1 }</pre>

```

665     }
666     \tl_if_empty:NTF ##2 { ##4 } { ##3 }
667   }
668   \switch:nNTF 5 \l_a_tl {
Single expression
669     \bool_set_false:N \l_no_range_bool
670     \__beanover_append:VN \l_a_tl \l_ans_tl
671     \seq_put_right:NV #1 \l_ans_tl
672   } {
673     \switch:nNTF 2 \l_a_tl {
674       \switch:nNTF 4 \l_b_tl {
675         \switch:nNTF 3 \l_a_tl {
Single expression
676           \__beanover_append:VN \l_a_tl \l_ans_tl
677           \tl_put_right:Nn \l_ans_tl { - }
678           \__beanover_append:VN \l_b_tl \l_ans_tl
679           \seq_put_right:NV #1 \l_ans_tl
680         } {
Single expression
681           \__beanover_append:VN \l_a_tl \l_ans_tl
682           \tl_put_right:Nx \l_ans_tl { - }
683           \tl_put_right:Nx \l_a_tl { - ( \l_b_tl ) + 1 }
684           \__beanover_append:VN \l_b_tl \l_ans_tl
685           \seq_put_right:NV #1 \l_ans_tl
686         }
687       } {
Single expression
688         \__beanover_append:VN \l_a_tl \l_ans_tl
689         \tl_put_right:Nn \l_ans_tl { - }
690         \seq_put_right:NV #1 \l_ans_tl
691       }
692     } {
693       \switch:nNTF 4 \l_b_tl {
694         \switch:nNTF 3 \l_a_tl {
Single expression
695           \tl_put_right:Nn \l_ans_tl { - }
696           \__beanover_append:VN \l_a_tl \l_ans_tl
697           \seq_put_right:NV #1 \l_ans_tl
698         } {
699           \msg_error:nnx { beanover } { :n } { Syntax-error(Missing-first):~#1 }
700         }
701       } {
Single expression
702         \seq_put_right:Nn #2 { - }
703       }
704     }
705   }
706 } {

```

Error

```
707 \msg_error:nnn { beanover } { :n } { Syntax~error:~#1 }
708 }
709 }
```

---

\\_\_beanover\_eval:nN \\_\_beanover\_eval:nN {<overlay query list>} <tl variable>

---

Evaluates the comma separated list of <overlay query>'s, replacing all the named overlay specifications and integer expressions by their static counterparts by calling \\_\_beanover\_eval\_query:nN, then append the result to the right of the <tl variable>. This is executed within a local group. Below are local variables and constants used throughout the body of this function.

\l\_query\_seq Storage for a sequence of <query>'s obtained by splitting a comma separated list.

(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.

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

(End definition for \c\_\_beanover\_comma\_regex.)

No other variable is used.

```
711 \cs_new:Npn \__beanover_eval:nN #1 #2 {
712   \group_begin:
```

Local variables declaration

```
713 \tl_clear:N \l_a_tl
714 \tl_clear:N \l_b_tl
715 \tl_clear:N \l_ans_tl
716 \seq_clear:N \l_ans_seq
717 \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 <first>::<last> range expressions as well. We first split the expression around commas, into \l\_query\_seq.

```
718 \__beanover_append:nN { #1 } \l_ans_tl
719 \exp_args:NNV
720 \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.

```
721 \seq_map_tokens:Nn \l_query_seq {
722   \__beanover_eval_query:Nn \l_ans_seq
723 }
```

We have managed all the comma separated components, we collect them back and append them to <tl variable>.

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

```

727 }
728 \cs_generate_variant:Nn \__beanover_eval:nN { VN, xN }

```

---

**\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:nN within a group. \l\_ans\_tl is used locally to store the result.

```

729 \NewExpandableDocumentCommand \BeanoverEval { s o m } {
730   \group_begin:
731   \tl_clear:N \l_ans_tl
732   \IfBooleanTF { #1 } {
733     \bool_set_true:N \l_no_counter_bool
734   } {
735     \bool_set_false:N \l_no_counter_bool
736   }
737   \__beanover_eval:nN { #3 } \l_ans_tl
738   \IfValueTF { #2 } {
739     \exp_args:NNNV
740     \group_end:
741     \tl_set:Nn #2 \l_ans_tl
742   } {
743     \exp_args:NV
744     \group_end: \l_ans_tl
745   }
746 }

```

### 6.3.7 Reseting slide ranges

---

**\BeanoverReset** [*<first value>*] {*<Slide list name>*}

---

```

747 \NewDocumentCommand \BeanoverReset { O{1} m } {
748   \__beanover_reset:nn { #1 } { #2 }
749   \ignorespaces
750 }

```

Forwards to \\_\_beanover\_reset:nn.

---

**\\_\_beanover\_reset:nn** {*<first value>*} {*<slide list name>*}

---

Reset the counter to the given *<first value>*. Clean the cached values also (not usefull).

```

751 \cs_new:Npn \__beanover_reset:nn #1 #2 {
752   \bool_if:nTF {
753     \__beanover_if_in_p:n { #2/A } || \__beanover_if_in_p:n { #2/Z }
754   } {
755     \__beanover_gremove:n { #2/C }
756     \__beanover_gremove:n { #2//A }
757     \__beanover_gremove:n { #2//L }
758     \__beanover_gremove:n { #2//Z }
759     \__beanover_gremove:n { #2//N }
760     \__beanover_gput:nn { #2/C0 } { #1 }

```

```

761   } {
762     \msg_warning:nnn { beanover } { :n } { Unknown~name:~#2 }
763   }
764 }
765 \makeatother
766 \ExplSyntaxOff
767 \</package>

```