Enhanced BEAMER increments: the `beamincr` package

Maneesh Sahani

November 11, 2023

The `beamincr` package extends and enhances the incremental overlay mechanisms implemented in the BEAMER class. These include labels to refer to and manipulate overlay steps, an extended action syntax, and new increment-aware environments.

1 Background: overlays and increments

The basic BEAMER display unit is the `frame`. A frame may be rendered step-by-step, in which case the individual versions of the frame are called "overlays" or "slides". We will use these terms interchangeably. BEAMER allows you to place material on an arbitrary slide in a frame like this

Example:

```
\begin{frame}
  text on slides 1 and up\\
  \onslide<2->
  text on slides 2 and up\\
  \onslide<3-4>{
  text only on slides 3 and 4\\
  }
  \only<5>{text only on slide 5\\}
  more text on slides 2 and up\\
\end{frame}
```

You can read about the differences between `\onslide` and `\only`, and the many other overlay-sensitive commands, in the BEAMER user guide. Note in particular the difference between the argument form and the declaration forms of `\onslide`. `\only` only works with an argument.

This explicit numbering approach becomes burdensome when you want many overlays. You have to keep track of the numbers explicitly, and if you subsequently add a step early in the sequence you need to re-number the rest. Thus, BEAMER also provides an incremental overlay specification. The following code will produce the same effect as that above.

Example:

```
\begin{frame}
  \resetincr % not standard BEAMER
  text on slides 1+\\
  \onslide<+->
  text on slides 2+\\
  \onslide<+-+(1)>{ % increments counter by 1, despite the two +s
  text only on slides 3-4\\
  }
  \onslide<+>{} % increment counter by another
  \only<+>{text only on slide 5\\}
  more text on slides 2+\\
\end{frame}
```

This form allows easy automation using default overlay specifications. For instance (from the BEAMER user guide)

Example:

```
\begin{itemize}[<+-| alert@+>]
\item Apple
\item Peach
\item Plum
\item Orange
\end{itemize}
```

There are important and sometimes not-entirely-intuitive differences between the incremental and explicit numbering systems. So we will refer to the steps implied in this way as "increments". They will mostly match slide numbers, but not always, as this example shows:

Example:

```
\begin{frame}
  \resetincr % not standard BEAMER
  text on slide 1+\\
  \onslide<3>{text on slide 3}\\
  text on slide 1+\\
  \onslide<+->
  text on slide 2+\\
  \onslide<4->
  text on slide 4+\\ % increment number is still 2!
  \onslide<+->
  text on slide 3+\\
\end{frame}
```

The increments have their own internal logic (specifically, their own internal counter) which is not affected by any explicit slide specifications that may appear between incremental calls. It may make sense to think of the increment number as being associated with *where* in the source file the material appears, rather than (necessarily) on *which slide* it appears.

There are a couple of oddities with the way increments work that often trip up first-time users. There are also some extensions that would be nice, like the ability to refer to a specific increment elsewhere in the frame. These things are certainly possible in stock BEAMER, but take some digging into internals. The tools here make things a bit easier.

As an aside, BEAMER has another incremental overlay system based on the `\pause` command. This uses the same counter as increments (in fact, the counter is called `beamerpauses`), but inteprets it slightly differently. This difference is discussed in Section 9. As a result, the two sets of specifications don't play very well together, at least from the viewpoint of non-experts. More on this below. I strongly suggest avoiding `\pause` entirely when using `beamincr`.

2 Setting increments

`\resetincr`[⟨*incrnum*⟩]

Reset the increment number to 1, or to the value defined by the optional argument if given. This doesn't directly affect the slide on which any following text appears, but it does alter the effect of subsequent `<+>` or `<.>` increments (as well as the /!/ reference discussed below). This command may be useful to synchronise overlays in (say) two columns or between highlighted bullet points and highlighting in a figure.

Example:

```
\begin{frame}
  \resetincr
  \begin{center}
    Two lists \onslide<+->{in sync}
  \end{center}
  \begin{columns}
    \begin{column}{.2\textwidth}
      \begin{itemize}[<+-| alert@+>]
      \item Apple \item Peach \item Plum \item Orange
```

```
      \end{itemize}
    \end{column}
    \begin{column}{.2\textwidth}
      \resetincr[2] % restart the increment counter to sync
      \begin{itemize}[<+-| alert@+>]
      \item green \item yellow \item purple \item orange
      \end{itemize}
    \end{column}
  \end{columns}
\end{frame}
```

Any optional argument must either be a number or be an increment reference enclosed in `//` (these are defined in Section 3). It cannot specify any sort of range, or be `+` or `.`, although `/./` and things like `/.(2)/` are allowed.

It is useful to call `\resetincr` at the start of every increment-based slide (as we have in the examples here). This avoids some potentially confusing behaviour that comes from the way the increment counter is implemented in BEAMER:

Example:

```
\begin{frame}
  text on slides 1-\\
  \onslide<+->
  text still on slides 1-\\
  \onslide<+->
  text on slides 2-
  \resetincr\onslide<.->
  text on slides 1-\\
  \onslide<+->
  text on slides 2-
\end{frame}
```

The first call to `\onslide<+->` doesn't advance the slide, unless it has been preceded by a `\resetincr` (or another `\onslide<+->` or a `\pause`).

`\fromincr<`⟨*incrnum*⟩`>`

This is shorthand for

```
\resetincr[incr]
\onslide<.->
```

It can only be used as a declaration (not with an argument). The restrictions on ⟨*incr*⟩ are the same as above.

3 Labelling and referring to increments

In complicated frames, it may be useful to name certain increments for reference elsewhere. For instance, one might want to change a figure at certain steps while progressing through a list of bullet points. Or one might want to redisplay certain slides in the frame with `\againframe` or `\handoutframe` (described below).

`\incrlabel<`⟨*incrnum*⟩`>{`⟨*label*⟩`}`

`\incrlabel<`⟨*incrnum*⟩`>`⟨`=`⟩`...//`⟨*label*⟩`/`

By default, this command attaches the current increment number to the label ⟨*label*⟩. Once defined, the labelled increment can be recovered in (almost) any overlay spec using the constructs discussed below. The ⟨*label*⟩ can contain most characters, but should not start with `=` or contain any of `()-` .

The `=` in the second form is optional, but if it is present then the `/`⟨*label*⟩`/` may be separated from the `=` by additional material, which will be left in place. The label must appear at the same grouping level as the `\incrlabel=` command and before the end of the current paragraph. This is similar to the behaviour of the `=` action described in Section 4.2.

If the optional ⟨*incr*⟩ is provided, ⟨*label*⟩ is set to its value. The restrictions on ⟨*incr*⟩ are the same as for `\resetincr`: it can be a number or an increment specification. This allows forms like

`\incrlabel</.(2)/>x` to set `x` to the current increment + 2. See the discussion of increment specifications below.

If `{`⟨*label*⟩`}` starts with a number in parentheses (e.g. `(2)x`) then this number is added to the current increment, or to the value of ⟨*incr*⟩, to obtain the label value. Thus, the effect of the command above can also be achieved by `\incrlabel{(2)x}`.

`\incrref{`⟨*incrref*⟩`}`

This command returns the increment number defined by increment reference ⟨*incrref*⟩ as described below.

The general form of an increment reference is

⟨*incrref*⟩: ⟨*label*⟩`(`⟨*offset*⟩`)`

The label can be a string assigned by a call to `\incrlabel`, or be one of the following special characters:

`.` The current increment (roughly equivalent to the incremental overlay specification '`.`', but can be used in places where only an ⟨*incrref*⟩ is valid).

`^` The first overlay used in the frame. This will usually evaluate to 1, but could be different if a slide range is specified for the frame. It can be used to display text on the first slide shown, even if this is not slide 1. For labelled frames (used with `\againframe`), the definition is tied to the label, and so will evaluate to the first overlay shown so far across all the uses of the frame.

`$` The last overlay (used) in the frame. This is stored in the `.aux` file and so generally will only be correct after a second compilation. It is tied to the frame label and so can be used to display only the final slide of a frame again: `\againframe</$/>{`⟨*frame label*⟩`}`. By combining with `^` it is possible to set a slide range at the first use and then automatically follow the same range with subsequent `\againframe` calls.

Example:

```
\begin{frame}<2-5>[label=myframe] % only use slides 2-5 in this presentation
  ...
\end{frame}
...
\againframe</^/-/$/>{myframe} % show slides 2-5 again
...
\againframe</$/>{myframe} % show only slide 5
```

`$*` The highest numbered overlay in the frame used so far. This may be useful to have an `\againframe` command continue to display a partially rendered frame, starting from the latest overlay reached in any previous displays.

`@` The most recently displayed overlay number in the frame. This may be useful to continue display picking up from where the last display left off.

`!` The increment number set by the most recent `\resetincr` command (or reset action as introduced in Section 4.1).

The ⟨*offset*⟩, if given, is added to the increment indicated by the label. It can be negative.

Increment references can be used as part of almost any overlay specification by enclosing them within slashes, e.g. `</foo(2)/>`.

Example:

```
\begin{frame}[label=twolists]
  \resetincr
  \begin{center}
    Two lists \onslide<+->{in sync}\\
    \onslide<+->{with more material\\}
    \onslide<+->{at the top}
  \end{center}
  \begin{columns}
```

```
    \begin{column}{.2\textwidth}
      \incrlabel{startlist}%
      \begin{itemize}[<+-| alert@+>]
      \item Apple \item Peach \incrlabel{halfway} \item Plum \item Orange
      \end{itemize}
    \end{column}
    \begin{column}{.2\textwidth}
      \resetincr[/startlist/]% keep in sync, even if we add extra topmatter
      \begin{itemize}[<+-| alert@+>]
      \item green \item yellow \item purple \item orange
      \end{itemize}
    \end{column}
  \end{columns}
  \vfill
  \onslide<+->
  The final increment is \incrref{.}.
  \incrlabel{end}
\end{frame}
```

Note that of commands discussed here, `\incrref` expects an ⟨*incrref*⟩ specification (i.e., ⟨*label*⟩`(`⟨*offset*⟩`)`), while `\resetincr`, `\fromincr` and `\incrlabel` expect an ⟨*incrnum*⟩ specification that might be an ⟨*incrref*⟩ in `//` (i.e., `/`⟨*label*⟩`(`⟨*offset*⟩`)/`) or just a number. Standard overlay-aware commands should all accept overlay specifications that include ⟨*incrnum*⟩s.

One BEAMER command (slightly patched in this package) with which named increments are particularly useful is `\againframe`. So

Example:

```
\againframe<1,/halfway/,/end(-1)/-/end/>{twolists}
```

provides an abbreviated tour of the lists. Increment labels are associated with the label of the enclosing frame, and so the same names can safely be reused across multiple named frames.

There is also a similar new command called `\handoutframe` to render more than one overlay from a frame in `handout` or similar modes that otherwise just show a single slide with all the overlays collapsed. See Section 8.

4 Enhanced overlay action specifications

This section discusses further extensions to the overlay specification syntax, and its interaction with increments and increment labels. Many of these extensions are only valid in a context that supports BEAMER actions. According to the user guide, these are `\action`, `\item`, the `actionenv` environment and block environments like `block` and `theorem`. This package adds the fields of incremental (Section 5) and incremental alignment (Section 6) environments to this list. In the absence of any action specifications, `\action` acts like `\uncover`.

4.1 Setting increments in overlay action specifications

`<resetincr@`⟨*incrnum*⟩`>`

`<`⟨*incrnum*⟩`-!`⟨*incrnum*⟩`-`⟨*incrnum*⟩`>`

The current increment number can be reset using either the explicit `resetincr@`⟨*incrnum*⟩ action, or an implicit `<!`⟨*incrnum*⟩`>` specification. The `!`⟨*incrnum*⟩ may appear at either the start or end of a (possibly open) range, so no more than one of the optional ⟨*incrnum*⟩s should be present. To set to an intermediate value, use a specification like `<1-4,!3>`.

Example:

```
\action<3-|resetincr@3>{body}
\action<!3->{body}
```

The increment number can be a label, with optional offset:

```
\incrlabel<2>{x}
\resetincr
\action</x/->{body on 2+}
\onslide<.->{this on 1+}
\action<!/x(2)/->{body on 4+}
\onslide<+->{this on 5+}
```

The forms `<!+>` and `<!.>` aren't supported (and wouldn't be useful: `<+>` already advances the increment, while `<!.>` would set it to its current value). However `<!/.(`⟨*offset*⟩`)/>` (note the `/` label notation) can be used to advance the increment counter by multiple (or negative) steps.

The reset takes effect after the overlay specification has been interpreted and before the body is set. So any `+` or `.` references will be relative to the increment in effect *before* the `\action`. However, the special increment label `/!/` can be used to access the most recent reset (but note the discussion of evaluation order below).

Example:

```
\resetincr
\action<!/.(2)/-|alert@.>{alerts too early}
\action<!/.(2)/->{\alert<.>{alerts when uncovered}}
\action<!/.(2)/-|alert@/!/>{alerts when uncovered}
```

It is possible to issue multiple implicit and explicit resetincr action commands in one overlay spec (including at both limits of a ⟨*range*⟩). The increment number in effect after the ⟨*action spec*⟩ will be determined by the first explicit `resetincr@` action, or if there are none, the first implicit `!`⟨*incrnum*⟩ specification.

If any actions within the same ⟨*action spec*⟩ depend on a `/!/` increment reference, then this value will be determined by the *following* `resetincr@` value, or if there are none, the *first* `!`⟨*incrnum*⟩, or if there are none, the most recent increment reset (by a `\resetincr` command or an action). If there are no preceding resets in the frame, then `/!/` (along with any offset) evaluates to 0. See the discussion of `\allowundefinedincrlabels` for more on 0-valued references, but note that this behaviour for `/!/` is independent of the `\allowundefinedincrlabels` state.

Example:

```
\resetincr
\action<1-5|resetincr@3|alert@/!/>{alerts on slide 1\\}
\action<1-5|alert@/!/>{alerts on slide 3\\}
\resetincr
\action<1-5|alert@/!/|resetincr@3>{alerts on slide 3\\}
\action<1-!5|point@/!/|resetincr@4|alert@/!/>{points on slide 4; alerts on slide 5\\}
```

This ordering behaviour can be understood from the following two facts: BEAMER applies actions in the reverse order to that in which they appear in the specification, and `beamincr` maps `!`⟨*incrnum*⟩ specifications to explicit actions at the *end* of the ⟨*action spec*⟩, preserving their order.

Actions must be used with argument text (usually enclosed in braces) or as environments. There is no equivalent to the declaration form of `\onslide`. Note, however, that `\fromincr` (Section 2) implements an `\onslide` declaration while also setting the current increment. See also the `incremental` environment (Section 5.1).

4.2 Assigning labels from overlay action specifications

`<···|=(`⟨*offset*⟩`)|···>{.../(`⟨*offset*⟩`)`⟨*label*⟩`/ ...}`

This syntax can be used to assign a label using an action specification. The name of the label to be assigned must be enclosed in `//`s within the *argument* of the `\action` (or `\item` or `\next` or alignment field ...)[1] The label is assigned to the increment number after any `+` or `!` actions have been interpreted, as though it was called with `\incrlabel` in place. Thus in this code

Example:

```
\resetincr[3]
```

[1] BEAMER actions don't make it possible to pass a text argument to the handler.

```
\action<!/.(2)/>{\incrlabel{x}action text}
\resetincr[3]
\action<!/.(2)/|=>{/x/action text}
\resetincr[3]
\action<!/.(3)/>{\incrlabel</.(-1)/>{x}action text}
\resetincr[3]
\action<!/.(3)/|=(-1)>{/x/action text}
\resetincr[3]
\action<!/.(3)/|=>{/(-1)x/action text}
```

the first two action calls set the label `x` to 5. The last three illustrate the use of assignment offsets: if `=` is followed by a number in parentheses, this is treated as an offset to add to the current increment at assignment, in the same way as indicated by the optional `<`⟨*incrnum*⟩`>` argument to `\incrlabel`. The same effect can be achieved by placing the offset before the label name within the enclosing `//`.

If no `/`⟨*label*⟩`/` is found or if ⟨*label*⟩ is empty, the action tries to do nothing quietly. This makes it possible to use an `=` in a default spec, while only assigning a label on selected steps. However, this behaviour comes with warnings, and should be used with caution. First, because of the way BEAMER's internals work, it is not currently possible to omit the `//` in an `\item`, although the label can be empty (omission is fine in the fields of an incremental or incremental alignment environment). Second, if there happens to be one more or more `/` characters in the argument to the action, the text between them (or from a single `/` to the end) will be interpreted as a label, unless they appear in a group within the argument, or an explicit `//` pair appears first. You have been warned!

4.3 Extending default overlay or action specifications

`<···|~|···>`

Ordinarily, explicit overlay or action specifications override any defaults that might apply. It may sometimes be convenient to instead extend the default. The `~` spec can be used to add the current default spec fields into an explicit overlay specification. In this form, the `~` may be proceeded by, say, a mode specification, but must not be followed by any text with the specification field (i.e., to the next `|` or `>`).

Example:

```
\begin{itemize}[<+-| alert@+|=>]
\item/ap/ Apple \item/pe/ Peach \item/pl/ Plum \item/or/ Orange
\end{itemize}
\begin{itemize}[<alert@/!/>]
\item<!/pe/-|~> yellow \item<!/or/-|~> orange \item<!/ap/-|~>green \item<!/pl/-|~> purple
\end{itemize}
```

Within incremental alignment environments (Section 6), a `~` will incorporate the field-specific default.

This extension is available is any overlay specification.

`<defaultspec@`⟨*range*⟩`>`

Action to set the default specification for overlay references within the action argument. The ⟨*range*⟩ may be any valid specification for a set of slides (such as `1`, `+-` or `!/foo/-/bar/`). The range is evaluated within the context of the `defaultspec@` specification, yielding specific slide numbers. Thus, any labels (including `/./` and `/!/`) or `+` or `.` symbols will be replaced by their current values. For example, in

```
\resetincr[3]
\action<+-|defaultspec@+>{\only<~>{only body} other stuff}
```

the specification to `\only` is set to `<4>`, not to `<+>`, which would evaluate to slide 5 in context. See also the `~` action below.

`<~`⟨*range*⟩`|···>`

This is equivalent to `<`⟨*range*⟩`|defaultspec@`⟨*range*⟩`|···>`. That is, it executes the calling action on ⟨*range*⟩ (using normal overlay evaluation rules) and also sets the default specification within the action argument to the evaluated range. This extension only works in an action specification context. As the

body of the argument will usually only be visible for the specified range anyway, the specification is most useful to control side effects.

Example:

```
\resetincr
\setcounter{displayed}{0}
\begin{itemize}[<+->]
\item \only<~>{\stepcounter{displayed}} % \only<+-> adds another increment
\item<~+-> \only<~>{\stepcounter{displayed}} % \only<4-> does not
\item<~~> \only<~>{\stepcounter{displayed}} % \only<5-> does not
\end{itemize}
```

In the example, the `\only` commands are used to advance the counter on slides where the `\item`s are visible. In both cases the overlay specification to `\only` is a copy of the surrounding default specification. For the first one, this is `<+->`. For the second, it is `<+->` *evaluated* within the `\item` call, giving `<3->`. The final case expands the second `~` to the default `+-`, which becomes the increment specification for the item, and then sets the default within the item to its value, which is `5-`.

4.4 Advanced references: using labels defined later

`\allowundefinedincrlabels[`⟨*flag*⟩`]`

If called alone, or with option ⟨*flag*⟩ > 0, tells LaTeX not to generate an error when encountering an undefined increment label. References to such labels instead evaluate to 0, and any offset in the reference is ignored. If ⟨*flag*⟩=0, the default error-generating behaviour is restored.

If a referenced label is defined later in the same frame, then it will take on that later-defined value on subsequent slides of the frame. Thus, in effect, this option makes it possible to refer to increment labels before they are defined. (Although material intended to be set on slide 1 cannot depend on such advance references.)

If the label is used as part of an open range then it may be necessary to use a special syntax in which the range indicator is placed *within* the `/ /` enclosing the label. If the label is undefined (and so 0), this syntax sets the other limit of the range to 0 as well.

Example:

```
% /foo/ is not defined on first evaluation
\onslide</foo/->{spec expands to <0->, so text appears on all slides}
\onslide</foo-/>{spec expands to <0-0>, so text is suppressed}
\incrlabel<2>{foo} % on later evaluations, both specs will expand to <2->
```

If the range is closed with an explicit numerical or (defined) label upper limit, then there is no current way to suppress early expansion. However forms like `/foo/-/foo(2)/` will evaluate to `0-0` as offsets are ignored for undefined labels.

Many problems with advanced references (including range expansion and the rendering of first-slide material) can be resolved by use of `\framescanonly` and `\againframe` (Section 8).

An `\allowundefinedincrlabels` command also makes it possible to *set* the current increment to an (initially) undefined label value using `\resetincr`, `\fromincr`, or `<!/label/>`, thereby setting the current increment to 0. Text set on that increment will not appear until the label is defined. However, any subsequent `<+>` specs will still advance the increment number, which may not be desired. This behaviour can be avoided by using the form `<!/.(1)/>` instead of `<+>`. The current increment label `/./` is treated in the same way as an undefined one when the increment is 0, and so the offset is ignored.

Example:

```
\resetincr{/foo/} % no list items appear until /foo/ is defined
\begin{itemize}<!/.(1)-/|alert@/!/>
\item foo
\item bar
...
\end{itemize}
```

If an initially undefined label is used to set the increment counter early in the frame, then increment labels that are defined later in the frame may change value once that first label is defined. This can be used for powerful effects, in which overlays in two different sections of the frame each depend on increments from the other. However, if the definition label used in the early reference is itself altered by the change in that early evaluation, then there is a risk of creating an infinite loop.

Example:

```
\begin{itemize}[<alert@/!/>]
\item<!/./-     |~|=(1)>/ping/ Apple  % 1- (!/./ ensures alert occurs on same slide)
\item<!/pong-/  |~> Peach             % 4-
\item<!/.(1)-/  |~|=>/ping2/ Plum     % 5-
\item<!/pong2-/ |~> Orange            % 8-
\end{itemize}

\begin{itemize}[<alert@/!/>]
\item<!/ping-/  |~> green             % 2-
\item<!/.(1)-/  |~|=(1)>/pong/ yellow % 3-
\item<!/ping2-/ |~> purple            % 6-
\item<!/.(1)-/  |~|=(1)>/pong2/orange % 7-
\end{itemize}
```

5 Incremental environments

The `beamincr` package provides some new increment-aware environments. These are described in the present section. It also makes it possible to use increment specifications within alignment environments such as `tabular` or $\mathcal{AMS}$-TEX `align`; these are discussed in Section 6.

Each new environment is accessible under two, otherwise equivalent, names. A common base name is either preceded by the word `incremental` or followed by the symbols `<>`. The environments in this section separate material into overlays using the token `\next` or `\next*`. Many will also apply an implicit or explicitly defined command to that material when `\next` is used, but omit the command for `\next*`.

Each environment makes it possible to specify a default overlay specification, applied to all `\next` fields, unless overridden by a local value. If no default is specified it is taken to be `<+->` (rather than any enclosing default), since it is assumed that `incremental` environments are intended to be, well, incremental.

5.1 Standard incremental environments

`\begin{incremental}[<`⟨*default specification*⟩`>]`
  `<`⟨*pre-next specification*⟩`>` ⟨*pre-next contents*⟩
  `\next<`⟨*next specification*⟩`>`
    ⟨*next contents*⟩
  `\next<`⟨*next specification*⟩`>`
    ⟨*next contents*⟩
      ⋮
`\end{incremental}`

This environment can be thought of as an increment-aware `itemize` without the list formatting. This makes it suitable for incremental control of a wider range of types of code, such as Ti*k*Z drawing commands. The keyword `\next` within the environment acts like `\item` in terms of incremental processing: the ⟨*next contents*⟩ are set within an `\action` command. Each `\next` call can be followed by an optional `<`⟨*next specification*⟩`>`, which is applied to the ⟨*next contents*⟩. If the specification is omitted, then the environment ⟨*default specification*⟩ is applied. If no ⟨*default specification*⟩ was given in the environment, then the default is inherited from the frame or container default. This is ordinarily `<*>`.

Unlike in `itemize` environments, code can also appear before the first `\next`. If any does, it is processed with the action specification given by `<`⟨*pre-next specification*⟩`>` if present, or else a default specification of `<.->` (to use the default specification defined for the environment, set `<`⟨*pre-next specification*⟩`>` to `<~>`). On the other hand, if nothing but whitespace appears between the opening of the environment—or the optional default overlay spec—and the first `\next`, then no action is applied.

A counter called `next` is set to 0 in the pre-next field and then advanced at every `\next`. An increment label called `next` ⟨*n*⟩, where ⟨*n*⟩ is the value of the `next` counter, is defined immediately after each `\next`, before any contents are processed. The label contains a (single) space before the number.

Example:

```
\resetincr
\begin{incremental}[<+->]
  <.-> this text on slide \incrref{next 0} (=1);
\next
  on slide \incrref{next 1} (=2);
\next<!/.(2)/->
  on slide \incrref{next 2} (=4), after \thenext\ (=2) next commands.
\end{incremental}
```

`\begin{<>}[<`⟨*default specification*⟩`>]`
⟨*environment contents*⟩
`\end{<>}`

This is a synonym for `\begin{incremental} ... \end{incremental}`.

The following environments apply a specified command to each overlay contents.

`\begin{incrementaldo} {`⟨*command definition*⟩`}[<`⟨*default specification*⟩`>]`
`<`⟨*pre-next specification*⟩`>` ⟨*pre-next contents*⟩
`\next<`⟨*next specification*⟩`>`
⟨*next contents*⟩
`\next<`⟨*next specification*⟩`>`
⟨*next contents*⟩
⋮
`\end{incrementaldo}`

This version treats each ⟨*next contents*⟩ as the argument to a command. The command takes a single argument and is defined by ⟨*command definition*⟩ in the same way as in `\newcommand`. As the command evaluation happens deep within the bowels of BEAMER processing, the single argument generally needs to be accessed as `####1`, *unless* the enclosing frame is declared to be `fragile` (in the frame options), in which case it is just `#1`.

The do command can be avoided for specific fields by using `\next*` in place of `\next`. Contents following `\next*` are set in the same way as in a plain `incremental` environment.

Any non-empty ⟨*pre-next contents*⟩ is always processed without the do command.

`\begin{do<>}{`⟨*command definition*⟩`}[<`⟨*default specification*⟩`>]`
⟨*environment contents*⟩
`\end{do<>}`

This is a synonym for `\begin{incrementaldo} ... \end{incrementaldo}`.

`\begin{incrementaldocmd}` `[`⟨*num args*⟩`]{`⟨*code*⟩`}[<`⟨*default specification*⟩`>]`
`<`⟨*pre-next specification*⟩`>` ⟨*pre-next contents*⟩
`\next<`⟨*next specification*⟩`>`
⟨*next contents*⟩
`\next<`⟨*next specification*⟩`>`
⟨*next contents*⟩
⋮
`\end{incrementaldocmd}`

This version inserts ⟨*code*⟩ (comprising arbitrary LaTeX commands) after each `\next` and before ⟨*next contents*⟩. If ⟨*code*⟩ is (or ends with) a command that takes one or more arguments, these will be read from ⟨*next contents*⟩. Braces may be needed within that text to delineate the arguments.

If the optional ⟨*num args*⟩ is non-zero, then this number of arguments is read from the text following `\next` and can be accessed using argument parameters (almost) as in `\newcommand`. As the command evaluation happens deep within the bowels of BEAMER processing, the parameter numbers must be protected with four `####` symbols, *unless* the frame is declared to be `fragile` (in the frame options) in which case a single `#` works.

Execution of the ⟨*code*⟩ can be avoided for specific fields by using `\next*` in place of `\next`. Contents following `\next*` are set in the same way as in a plain `incremental` environment.

Any non-empty ⟨*pre-next contents*⟩ is always processed without ⟨*code*⟩.

Example:

```
\tikz[every node/.style={above,allow upside down,midway,sloped}]{
  \begin{incrementaldocmd}[1]{\draw ({72*\thenext-72}:10ex)--(72*\thenext:10ex) node {#1};}
                          [<+-|alert@+|=>]
    \next/one/   {one}
    \next/two/   {two}
    \next/three/ {three (\thenext/\theincrement)}
    \next/four/  {four}
    \next/five/  {five}
    \next*/done/ \draw (0:5ex) \foreach \t in {1,...,5}{ -- (\t*72:5ex)};
                 \node [anchor=center] {done!};
  \end{incrementaldocmd}}
```

`\begin{docmd<>}` `[`⟨*num args*⟩`]{`⟨*code*⟩`}[<`⟨*default specification*⟩`>]`
⟨*environment contents*⟩
`\end{docmd<>}`

This is a synonym for `\begin{incrementaldocmd} ... \end{incrementaldocmd}`.

`\begin{incrementaldodef}` `[`⟨*parameter spec*⟩`]{`⟨*code*⟩`}[<`⟨*default specification*⟩`>]`
`<`⟨*pre-next specification*⟩`>` ⟨*pre-next contents*⟩
`\next<`⟨*next specification*⟩`>`
⟨*next contents*⟩
`\next<`⟨*next specification*⟩`>`
⟨*next contents*⟩
⋮
`\end{incrementaldodef}`

This is similar to the `incrementaldocmd` environment, but allows arguments to be specified using the flexible format of `\def`. Parameter numbers must be escaped with four `#`s in both specification and code, unless the frame is declared `fragile`. Code execution can be skipped using `\next*` and is always skipped for any ⟨*pre-next contents*⟩.

`\begin{dodef<>}` `[`⟨*parameter spec*⟩`]{`⟨*code*⟩`}[<`⟨*default specification*⟩`>]`
⟨*environment contents*⟩
`\end{dodef<>}`

This is a synonym for `\begin{incrementaldodef} ... \end{incrementaldodef}`.

`\begin{incrementaldolongdef}` `[`⟨*parameter spec*⟩`]{`⟨*code*⟩`}[<`⟨*default specification*⟩`>]`
⟨*environment contents*⟩
`\end{incrementaldolongdef}`

This form allows paragraph breaks within ⟨*next contents*⟩, but is otherwise the same as `\begin{incrementaldodef} ... \end{incrementaldodef}`,

`\begin{dolongdef<>}` `[`⟨*parameter spec*⟩`]{`⟨*code*⟩`}[<`⟨*default specification*⟩`>]`
⟨*environment contents*⟩
`\end{dolongdef<>}`

This is a synonym for `\begin{incrementaldolongdef} ... \end{incrementaldolongdef}`.

5.2 Ti*k*Z-based incremental environments

The following environments use Ti*k*Z picture commands to create their effects. They will only be defined if Ti*k*Z is also loaded in the document preamble.

```
\begin{incrementallayers} [⟨node options⟩][<⟨default specification⟩>]
    <⟨pre-next specification⟩> ⟨pre-next TikZ commands⟩
  \next<⟨next specification⟩>[⟨next node options⟩]
    ⟨layer contents⟩
  \next*<⟨next specification⟩>[⟨scope options⟩]
    ⟨TikZ commands⟩
      ⋮
\end{incrementallayers}
```

Set ⟨*next contents*⟩ within overlayed Ti*k*Z nodes. This may have the effect of later text appearing to be layered on top of earlier material. If the specifications place material on mutually exclusive increments (e.g. with `<+>`) then this has a similar effect to BEAMER's `overprint` environment. In particular, the region occupied by the environment will correspond to the largest of the nodes. However, this behaviour can be subverted by use of `\only` or similar commands within the ⟨*next contents*⟩, or `only@` actions in ⟨*next specification*⟩s; these should be used with care.

By default, nodes are set with `text width=\columnwidth, inner sep=0pt` so that the text fills the width of the current frame or column or minipage. They are aligned at their top borders. These defaults can be overridden, and arbitrary Ti*k*Z options provided to the nodes, in one of three ways: by setting options in the `incremental layer` Ti*k*Z style, by placing them in the optional ⟨*node options*⟩ argument to the environment, or by placing them in the optional ⟨*next node options*⟩ after a `\next`(and after any ⟨*next specification*⟩).

Three shorthand node alignment keys are available: `t` aligns the nodes at their top edges (the default); `b` at their bottom edges and `c` at their centres.

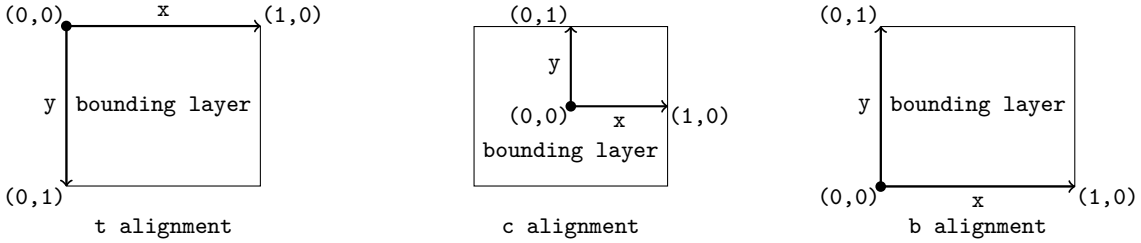
The entire environment is set within a single `tikzpicture`. Any ⟨*pre-next contents*⟩ and the contents of any `\next*` are interpreted as Ti*k*Z commands within that picture. Each `\next*` sets its contents within a Ti*k*Z `scope` environment. Options to this environment can be provided by starting the contents (after any ⟨*next specification*⟩) with `[`⟨*scope options*⟩`]`.

The default origin of the Ti*k*Z coordinate system is the `north`, `center` or `south` anchor points of the layer nodes for `t`, `c` and `b` alignment respectively. The nodes themselves are accessible under the names `(layer` ⟨*n*⟩`)`, where ⟨*n*⟩ is the corresponding value of the `next` counter.

The following special options can be given within `\next*` scope options.

`t` | `c` | `b` These options do not alter anything by themselves. However, subsequent calls to the options below will behave differently for different alignment keys. The default value is the alignment of the most recent layer node.

`layer xy` Set the `xy` coordinate system to span the bounding box of the layers rendered so far. The origin depends on the alignment of the most recent layer, or a `t` | `c` | `b` alignment option to the `\next*` scope. The origin is placed at the `north west`, `center` or `south west` of the bounding node, for `t`, `c`, or `b` alignment respectively. The `x` vector extends horizontally to the east border of the bounding node, while `y` extends vertically to the opposite side (or top for `c` alignment).



`layer grid` Draw a grid within the current `layer xy` coordinates. Fine grid lines are 0.1 units apart. Heavier lines are placed at full units.

\begin{layers<>} [⟨*parameter spec*⟩]{⟨*code*⟩}[<⟨*default specification*⟩>]
⟨*environment contents*⟩
\end{layers<>}

This is a synonym for \begin{incrementallayers} ... \end{incrementallayers}.

5.3 Synchronising incremental environments

After an incremental environment has been defined, the special label next+ evaluates to the increment associated with the next \next: or, more precisely, to the smallest /next ⟨*n*⟩/ that is strictly greater than the current increment or to 0 if none are larger. This label can be used to easily synchronise later incremental environments to a first one, even if each \next step involves multiple increments.

Example:

```
\begin{<>}
\next first line \onslide<+->{with an addendum}\\
\next second line\\
\end{<>}

\resetincr % ensure currentincr is less than /next 1/
\begin{<>}[<!/next +/->]
\next match the first line on slide \incrref{next 1}\\
\next match the second line on slide \incrref{next 2}\\
\next match the second line on slide \incrref{next +}\\
\end{<>}
```

The reset specification in <!/next +/-> is necessary to advance the current increment. The values of next ⟨*n*⟩ are redefined in the second environment, but mostly to the same values as in the first. The exception in this example is next 0 which will be set to 1 (by the \resetincr) regardless of its value in the first environment. As a side effect, this ensures that /next +/ at the first \next call evaluates to /next 1/ rather than /next 0/.

It is possible to wrap an itemize environment within an incremental one to provide a similar functionality, albeit with warnings discussed below.

Example:

```
\begin{<>}\begin{itemize}
\next\item<.-> first item \item with an addendum\\
\next\item<.-> second item\\
\end{itemize}\end{<>}

\resetincr[/next 0/] % ensure currentincr is less than /next 1/
\begin{itemize}[<!/next +/->]
\item match the first item on slide \incrref{next 1}\\
\item match the second item on slide \incrref{next 2}\\
\item match the second item on slide \incrref{next +}\\
\end{itemize}
```

The explicit <.-> specifications in the first itemize are needed because the incremental environment sets the default spec to <+->, and so would otherwise cause each \item to introduce a new increment. It is not possible to set separate default specs for the incremental and itemize environments when both are used explicitly. Thus, the addendum \item inherits the <+-> default.

The second itemize does not need to be wrapped within an incremental environment. It simply accesses the labels defined in the first.

Note that both the incremental and itemize environments group the contents of \next or \item commands within environments. it is thus surprising that the interleaved construct works at all. There are cases where it will fail. In general, when using \next\item it is best to have display (or uncover) control determined by the \next command, but appearance (such as alerts) determined by the \item.

A shorthand environment is available, which adds the capability to define separate default specifications for \next and \item calls. However, this does not resolve the fragility of the interleaving.

\begin{incrementalitemize}[<⟨*default specification*⟩>][<⟨*next item default*⟩>&<⟨*separate item default*⟩>]

```
    <⟨pre-next specification⟩> ⟨pre-next contents⟩
  \next<⟨next specification⟩>
    ⟨next contents⟩
  \item<⟨item specification⟩>
    ⟨item contents⟩
      ⋮
\end{incrementalitemize}
```

This environment is similar to

```
\begin{docmd<>}[0]{\item<.->}
\begin{itemize}
\next<next specification> next contents
\item<item specification> item contents
...
\end{itemize}
\end{docmd<>}
```

Each `\next` creates an `\item` by default displayed from the increment in effect after the `\next`. It also increments the `next` counter and defines a corresponding `next` ⟨*n*⟩ label. An explicit `\item` uses the same default specification as `\next` (which itself defaults to `<+->` in incremental environments), but does not set the next-related counter or label.

The second optional argument (which must follow the first) makes it possible to change the default overlay behaviour of the `\item`s. It can contain one or two specifications, separated by an & symbol. The first applies to the implicit `\item` commands generated by each `\next`. The second, if present, applies to any explicit `\item`s. If absent, it is set to the overall default specification. The default values if no optional arguments are given are `[<+->][<.->&<~>]`.

Note that any ⟨*pre-next contents*⟩ cannot generate typeset text, as this would precede the first `\item` command.

```
\begin{itemize<>}[<⟨default specification⟩>]
  ⟨environment contents⟩
\end{itemize<>}
```

This is a synonym for `\begin{incrementalitemize} ... \end{incrementalitemize}`.

6 Incremental alignment environments

Standard LaTeX alignment environments including `tabular` and the `align` and `align*` environments from the `amsmath` package are not ordinarily increment-aware. The current package introduces a partial fix for this, although their are remaining fragilities that may need to be worked around. It is possible to make an increment-aware version of any alignment environment using `\CreateIncrementalAlignmentEnvironment` as described below. A few such environments are defined automatically when `beamincr` is loaded and these are described first, thus illustrating the behaviour once an increment-aware environment has been created.

6.1 Automatically defined incremental alignment environments

The following two environments are equivalent:

```
\begin{incrementalalign*}[<⟨spec1⟩>&<⟨spec2⟩>&...]
  ⟨environment contents⟩
\end{incrementalalign*}

\begin{align*<>}[<⟨spec1⟩>&<⟨spec2⟩>&...]
  ⟨environment contents⟩
\end{align*<>}
```

Each pre-processes the input to `align*`, placing an `\action<>{}` command around each field, defined as the material appearing between successive `&`, `\\` or end environment tokens. By default, the first field

on a line is called with `\action<+->{}`, and up to 7 remaining ones with `\action<.->{}`. This has the effect of displaying a full line at a time, unless it has more than 8 fields. The optional argument makes it possible to change this behaviour to `\action<`⟨*spec1*⟩`>`, `\action<`⟨*spec2*⟩`>`, etc. with the sequence of specifications reset to `<`⟨*spec1*⟩`>` at the beginning of every line. If there are fewer specifications in the default than fields on a single line, then the sequence is repeated. The default specification values can be changed by calling `\setincrementalenvspec{align*}{<`⟨*new default*⟩`>}` or similar.

The default specification for a single field can be overridden by placing a field-specific specification in `<>` at its start. This means that a leading `<` in the field contents itself must be protected, e.g. by preceding it with `{}`.

The use of `\action` means that BEAMER will interpret both standard `action@`⟨*increment*⟩ actions and implicit ones such as `!`-prefixed increment resets, `=` label assignments or `~`⟨*range*⟩ default specification.

Example:

```
\begin{align*<>}[<+->&<.->] % increment after every two &s
  x\incrlabel{x} &= y  &  1 &{}< 2 \\
  </x/-> x^2 &= y^2    &  <!/x/-> e^{i\pi} &<+->= -1 \\
  \sum_n f(n) &<.-|alert@.> \to \int f(x) dx
\end{align*<>}
```

The pre-processor is not able to distinguish between the `&` alignment characters that apply to the containing environment and any that appear within enclosed environments, such as `array`. Thus, any such environments must be protected. The simplest approach is just to add extra braces to group the enclose environment at a lower level. Alternatively, the environment can be defined within a token register or a protected macro. It is still possible to use increments within the environments: these are processed sequentially with those in the containing `align` environment, respecting increment labels, resets etc.

Example:

```
% using grouping
\newtoks\mymatrix
\begin{align<>}
  \incrlabel{mat}{\begin{pmatrix} 1 & 2 \\ \alt<+->{3}{2} & 4 \\ \end{pmatrix}}\resetincr[/mat/]
  & \text{is \only<+->{not }singular}
\end{align<>}

% using \protected
\protected\def\mymatrix{\begin{pmatrix} 1 & 2 \\ \alt<+->{3}{2} & 4 \\ \end{pmatrix}}
\begin{align<>}
  \incrlabel{mat}\mymatrix \resetincr[/mat/]& \text{is \only<+->{not }singular}
\end{align<>}

% using token registers
\newtoks\mymatrix
\mymatrix={\begin{pmatrix} 1 & 2 \\ \alt<+->{3}{2} & 4 \\ \end{pmatrix}}
\begin{align<>}
  \incrlabel{mat}\the\mymatrix \resetincr[/mat/]& \text{is \only<+->{not }singular}
\end{align<>}
```

It may be wise to put any `\newtoks` declaration outside the frame so as not to consume more of TEX's resources than needed.

`\intertext` lines must be terminated with `\\`. By default they will be grouped within the action call of the last field of the preceding line. This behaviour can be changed by inserting a `\\` between that field and the `\intertext`. By default, both `\\`s will add extra vertical space (and an equation number in non-starred variants). These can be avoided by using a form like `\nonumber\\[-3ex]` instead.

The `amsmath` `\tag` command is processed in such a way that it cannot easily be made overlay aware. Any `\tag`s will appear on any slides where the overall environment is uncovered, even if no fields have appeared. However, an alternative `\eqtag` is defined. See Section 6.4.

`\begin{incrementalgather*}[<`⟨*default spec*⟩`>]`
⟨*environment contents*⟩

```
\end{incrementalgather*}

\begin{gather*<>}[<⟨default spec⟩>]
  ⟨environment contents⟩
\end{gather*<>}
```

These incremental forms are also created when `beamincr` is loaded, with similar behaviour to the incremental `align*` environments described above. Although these contain only one field per line, automatic access to BEAMER and `beamincr` actions as these lines are processed can be useful. By default, they uncover equations a line at a time (using `<+->`).

```
\begin{incrementaltabular}[⟨pos⟩]{⟨cols⟩}[<⟨spec1⟩>&<⟨spec2⟩>&...]
  ⟨environment contents⟩
\end{incrementaltabular}

\begin{tabular<>}[⟨pos⟩]{⟨cols⟩}[<⟨spec1⟩>&<⟨spec2⟩>&...]
  ⟨environment contents⟩
\end{tabular<>}
```

These provide increment-aware versions of the standard LaTeX `tabular` environment. By default, the entire table is uncovered on the current increment (using `<.->`), but this behaviour can be altered by changing the default specification when called, or by using `\setincrementalenvspec` as described below. It may also be desirable to uncover entries column-by-column. This effect can be achieved using increment labels.

Example:

```
\begin{tabular<>}{cccc}[</col1-/>&</col2-/>&</col3-/>&</col4-/>]
  <=(1)>/col1/\bf fruit & <=(2)>/col2/\bf colour
      & <=(3)>/col3/\bf climate & <=(4)>/col4/\bf family \\[1ex]
  Apple & green & cool & pome \\
  Peach & yellow & warm & drupe \\
  Plum & purple & cool & drupe \\
  Orange & orange & hot & citrus \\
\end{tabular<>}
```

```
\begin{incrementaltabular*}[⟨pos⟩]{⟨width⟩}{⟨cols⟩}[<⟨spec1⟩>&<⟨spec2⟩>& ...]
  ⟨environment contents⟩
\end{incrementaltabular*}

\begin{tabular*<>}[⟨pos⟩]{⟨width⟩}{⟨cols⟩}[<⟨spec1⟩>&<⟨spec2⟩>& ...]
  ⟨environment contents⟩
\end{tabular*<>}
```

These forms add the ⟨*width*⟩ argument of LaTeX's `tabular*` environment.

6.2 Creating new incremental alignment environments

```
\CreateIncrementalAlignmentEnvironment{⟨name⟩}[⟨Nopts⟩]{⟨Nreqs⟩}[<⟨default spec⟩>][⟨base⟩]
```

Create an increment-aware version of an alignment environment. Unless a different ⟨*base*⟩ environment is specified in the final argument, the new environment is based on an existing one of the same ⟨*name*⟩. This existing environment should process its contents in fields demarcated by `&` and/or `\\` tokens. The arguments ⟨*Nopts*⟩ and ⟨*Nreqs*⟩ specify the numbers of optional and required arguments the base environment expects. If ⟨*Nopts*⟩ is omitted it is taken to be 0. ⟨*Nreqs*⟩ must be specified, but can be 0. If the ⟨*default specification*⟩ is omitted it is set to `<.->`, thus displaying the environment contents at the prevailing increment number in the frame.

The new environment can be accessed using either of the names `incremental`⟨*name*⟩ or ⟨*name*⟩`<>`.

6.3 Manipulating default behaviour

\useincrementalenv{⟨*name*⟩}

Make all subsequent uses of the ⟨*name*⟩ environment call the incremental version. The incremental version must already have been created.

Example:

```
\useincrementalenv{align*}
\begin{align*}[<+->]
  % this is an incremental environment
\end{align*}
```

The specified ⟨*name*⟩ must match the *name* of the base environment (i.e., the first argument to `\CreateIncrementalAlignmentEnvironment`), whether or not this is the same as its base.

Example:

```
\CreateIncrementalAlignmentEnvironment{foo}{0}[<.->][bar]
\begin{foo}
  % error -- environment is accessible as incrementalfoo or foo<>
\end{foo}
\useincrementalenv{foo}
\begin{foo}
  % evokes the incremental version of bar
\end{foo}
```

\usenonincrementalenv{⟨*name*⟩}

Make subsequent uses of the ⟨*name*⟩ environment refer to the non-incremental version. If the name and base specified at creation were the same, this restores the normal behaviour of the ⟨*name*⟩ environment. If a different base environment was specified at creation, this creates a new ⟨*name*⟩ environment that is synonymous with the base.

Example:

```
\CreateIncrementalAlignmentEnvironment{foo}{0}[<.->][bar]
\begin{foo}
  % error -- environment is accessible as incrementalfoo or foo<>
\end{foo}
\usenonincrementalenv{foo}
\begin{foo}
  % evokes the original version of bar
\end{foo}
```

\setincrementalenvspec{⟨*name*⟩}{⟨*default specification*⟩}

Set the default specification for incremental environments of type ⟨*name*⟩.

6.4 Equation numbering

Unfortunately, the `amsmath` `\tag` command, used for equation numbering, is processed in such a way that it cannot easily be made overlay aware. Any `\tag`s will appear on any slides where the overall environment is uncovered, even if no fields have appeared. This is also the case with automatic numbering. Thus if defined using

```
\CreateIncrementalAlignmentEnvironment{gather}{0}[<+->]
```

the `gather<>` environment will also generate all equation numbers whenever the environment as a whole is uncovered, regardless of the status of the relevant fields. In principle, this behaviour could be partially addressed using a technique discussed in the BEAMER manual Howtos, but this requires some additional hackery.

Instead, beamincr provides an increment-aware version of `\tag` and of automatic equation numbering.

\eqtag<⟨*overlay spec*⟩>{⟨*tag*⟩}

Place ⟨*tag*⟩ on slides that match ⟨*overlay spec*⟩.

\eqnum<⟨*overlay spec*⟩>

Place the current equation number (as `\theequation`) on slides that match ⟨*overlay spec*⟩, and then increment it.

The second form can be used as an action.

<eqnum@⟨*range*⟩>

Place the current equation number (as `\theequation`) on slides in ⟨*range*⟩, and then increment it.

Example:

```
\begin{align*}[<+-|eqnum@+->&<.->]
  e^{\pi i} &= -1 \\
  \sqrt[3]{1} &= e^{2\pi i/3}
\end{align*}
```

numbers equation as it is uncovered.

In principle, this method can be used to create automatic numbering forms of the amsmath environments:

```
\CreateIncrementalAlignmentEnvironment{gather}{0}[<+-|eqnum@+->][gather*]
\begin{gather<>}
  % equations will be numbered, with numbers uncovered with the rest of the line
\end{gather<>}
```

Such environments are not created automatically, as the user should be aware of two traps. First, if a default specification is given to `\begin{gather<>}`, the environment will revert to unnumbered unless the appropriate `eqnum` action or commands are provided. Second, a call to `\usenonincrementalenv{gather}` will make `gather` a synonym for `gather*`. At this point, there would be no easy way to restore `gather` to its original behaviour. You have been warned.

7 Directing attention

7.1 Alerts

\alerts<⟨*overlay spec*⟩>{⟨*argument contents*⟩}

Activate any `\alert*` commands in ⟨*argument contents*⟩ on the overlays indicated by ⟨*overlay spec*⟩.

<alerts@⟨*range*⟩>

This is the action equivalent of `\alerts`.

\alert*{⟨*argument*⟩}

Executes `\alert{`⟨*argument*⟩`}` if an enclosing `\alerts` command or action is active. Otherwise ⟨*argument*⟩ is displayed unalerted.

7.2 Pointers

\point<⟨*overlay spec*⟩>{⟨*contents*⟩}

In normal text, insert a pointer before ⟨*contents*⟩ on the specified slides. If ⟨*contents*⟩ includes any `\point*` commands, pointers are inserted at the locations of these commands at the same time.

If called within a Ti*k*Z picture, the `\point` command does not insert a pointer itself. Instead, any pointer defined by the `pointer` option to any nodes in ⟨*contents*⟩ is activated. See the description of `pointer` below.

`<point@`⟨*range*⟩`>`

The action form of `\point` can be used in all contexts where an action specification is valid. Its behaviour around normal text or Ti*k*Z code is as above. In `itemize` environments it replaces the default item label with the pointer[2]. In `enumerate` environments it prepends the pointer to the default label. In other list environments, or when the label is set explicitly as an optional argument to `\item`, it has no direct effect. However any `\point*` commands in the item label or text are activated.

`<pointers@`⟨*range*⟩`>`

This action activates `\point*` commands in the argument contents, but does not insert a pointer.

`\point*[`⟨*options*⟩`]{`⟨*contents*⟩`}`

If the command is not followed by a `[` or `{` character, insert a pointer when an enclosing `\point` command or action is active.

If followed by an argument in `[]` or `{}`, and if Ti*k*Z is loaded, call `\pointtonode` as described below. If Ti*k*Z is *not* loaded a normal pointer is inserted as in the no-argument form, any options are ignored, and the contents is copied to the output.

`tikz option: pointer=[`⟨*pointer node options*⟩`]`⟨*angle*⟩

This is a Ti*k*Z option that can be passed to a `node` to insert a pointer drawn towards the node whenever an enclosing `\point` command or action is active. If an ⟨*angle*⟩ is specified, the pointer is drawn towards the corresponding point on the node boundary; this can be specified as a numerical angle or a direction like `north`. The default angle is `west` or `180`, so that the pointer points to the node from the left. The pointer itself is drawn within a node: this behaviour is very similar to the regular Ti*k*Z node `label` option, except that the pointer node is automatically sloped so as to point inwards.

The current implementation does not work well with `coordinate`s. The alternative `pointer coordinate` style creates an empty circular node of 0.1pt size, which is broadly equivalent.

If ⟨*pointer node options*⟩ are specified (generally requiring braces around the entire option value to protect Ti*k*Z's option parsing from seeing the `[]`s) these are passed to the pointer node. A few options may be particularly useful:

`pos=`⟨*scale*⟩ adjusts the placement of the pointer as a fraction of the distance from the target node centre to its boundary. The default is 1.0. This option is unlikely to be useful when the target is a `pointer coordinate`.

`pointer sep=`⟨*dimen*⟩ adds ⟨*dimen*⟩ to the distance of the pointer from the target node.

`rotate=`⟨*angle*⟩ rotates the pointer relative to its initial angle.

`\pointtonode[`⟨`pointer` *options*⟩`]{`⟨*contents*⟩`}`

If Ti*k*Z is loaded, this is shorthand for

```
\tikz[baseline]\node[anchor=base,text height=1.5ex,inner sep=0pt,pointer={#1}]{#2};
```

The spacing adjustments ensure that the contents in the node are printed in alignment with the surrounding text, and that pointers to an empty node appear at a similar height to those inserted by `\point` or `\point*`. The availability of ⟨`pointer` *options*⟩ provides flexibility in the pointer placement.

If Ti*k*Z is not loaded, this is the equivalent of `\point*{`⟨*contents*⟩`}`, ignoring any options given.

`\usepointer[`⟨*inactive glyph*⟩`]{`⟨*pointer glyph*⟩`}`

Use ⟨*pointer glyph*⟩ for subsequent pointers in the current group. If the optional argument is absent, then the pointer is replaced by a phantom of the same size when inactive (the size only matters if `\useuncoverpointer` is active). If it is given, then inactive pointers are replaced by ⟨*inactive glyph*⟩ (which may be empty).

Example:

```
\usepointer{\raisebox{0.3ex}{\alert{$\blacktriangleright$}}} % the default
\usepointer{\raisebox{-0.4ex}{\alert{\HandRight}}} % requires \usepackage{bbding}
```

[2] Although if the `itemize` is nested within an `enumerate`, it inherits the `enumerate` behaviour.

Note that some adjustment of the vertical placement, as in these examples, may be necessary to align the pointer appropriately with the text.

The effect of this command is local to the containing group.

The pointer appearance should really be controlled through BEAMER's template mechanism, but that's a project for another day.

\useoverprintpointer

Print subsequent pointers (and any inactive glyphs) in the current group *over* existing material, without reserving any space for them (internally, they set within a zero-width box). This is the default, and avoids the dilemma of either leaving blank spaces for inactive pointers, or having text rearrange when the pointer appears.

The effect of this command is local to the containing group.

\useuncoverpointer

Set subsequent pointers and any inactive glyph in the current group as normal text, taking up space on the page. If no inactive glyph has been specified, the effect is to leave a blank space when the pointer is inactive, much like the effect of the `\uncover` or `\onslide` commands. If the inactive glyph is set to the empty string, there is no blank space, but surrounding text is rearranged to make room for the pointer when it becomes active.

The effect of this command is local to the containing group.

7.3 Graphical alerts

If Ti*k*Z is loaded, `beamincr` defines various styles of graphical alerts or `glerts`, and provides a facility to define more. The argument to be alerted is set within an inline Ti*k*Z node with a tight bounding box, aligned with the baseline of the surrounding line of text. A second node or associated path is used to add the graphical embellishment. (More details of the construction are given in Section 7.3.7.) Glerts can be used to draw attention at specific increments, or simply to highlight material on the slide in a static fashion.

7.3.1 Glert commands

The following commands are used to insert and control the activation of glerts.

\useglert{⟨*style*⟩$_1$=⟨*options*⟩, ⟨*style*⟩$_2$=⟨*options*⟩, [⟨*further options*⟩]}

Set one or more glert styles to be applied to subsequent `\glert[*]` calls or actions, unless they define a style locally. For a discussion of the ⟨*further options*⟩, see Section 7.3.7.

\addglert{⟨*style*⟩$_1$=⟨*options*⟩, ⟨*style*⟩$_2$=⟨*options*⟩, [⟨*further options*⟩]}

Append one or more glert styles to the current set, to be applied to subsequent `\glert[*]` calls or actions, unless they define a style locally.

\glert<⟨*overlay spec*⟩>[⟨*options*⟩]{⟨*argument contents*⟩}

Apply a graphical alert to ⟨*argument contents*⟩ on ⟨*overlay spec*⟩. The style of the alert may be set in ⟨*options*⟩ using the same syntax as in `\useglert`. When inactive, ⟨*argument contents*⟩ is displayed unalerted within a Ti*k*Z node.

\glert*[⟨*options*⟩]{⟨*argument contents*⟩}

Executes `\glert`[⟨*options*⟩]{⟨*argument contents*⟩}, with the active or inactive state determind by the status of any enclosing `\glerts` command or action. If there is no such command, then the glert is always inactive.

`\glerts<`⟨*overlay spec*⟩`>{`⟨*argument contents*⟩`}`

Activate any `\glert*` commands within ⟨*argument contents*⟩ on the overlays indicated by ⟨*overlay spec*⟩.

`<glerts@`⟨*range*⟩`>`

This is the action equivalent of `\glerts`.

7.3.2 Glerts and modes

Ordinarily, glerts remain inactive in all except `beamer` mode. To include an active glert in (say) `handout` mode, include the mode spec `<handout>` or `<handout:*>` within the ⟨*spec*⟩ passed to `\glert`. The same options can can be included in the spec of an explicit `\glerts` command. To activate `glert*`s implicitly from an enclosing action, use `<handout:glerts@*>`.

Alternatively, when using `\handoutframe`, specific overlays can be included within the handout (or other mode) output rendered as they would be in `beamer` mode.

7.3.3 Built-in glert styles

A few `glert` styles are predefined. Each displays the text in the same style as the surrounding material when inactive. Their actions when activated are as follows.

`null` Do nothing, even when activated.

`alert=`⟨*color, ...*⟩ Display text in red or ⟨*color*⟩ (like the standard `\alert`). This is the default glert style.

`box=`⟨*color, ...*⟩ Draw a rectangular frame around the text in red or ⟨*color*⟩.

`ellipse=`⟨*color, ...*⟩ Draw an ellipse around the text in red or ⟨*color*⟩.

`uline=`⟨*line options*⟩ Underline the text. The appearance of the line can be altered with standard Ti*k*Z line options. The default is red and "very thick".

`highlight=`⟨*color, ...*⟩ Fill a background box with partly transparent red or ⟨*color*⟩, much as a highlighter would on paper.

`spotlight=`⟨*color, ...*⟩ Draw a fading ellipse underneath the text in partly transparent red or ⟨*color*⟩.

`connect={[`⟨*to options*⟩`]` ⟨*target*⟩`}` Draw a [red, very thick, double-headed arrow] line to the Ti*k*Z coordinate ⟨*target*⟩. This could be a `(`⟨*glert name*⟩`)` (see Section 7.3.4). If it is the name of a node in another Ti*k*Z picture, that picture must have the `remember picture` option set; this is always the case for glerts.

The ⟨*to options*⟩ can be used to modify the shape of the path. The style of the line can be changed by specifying `path={`⟨*options*⟩`}` within the ⟨*further options*⟩ of the `\glert[*]` command.

`callout={[`⟨*callout node options*⟩`]`⟨*angle*⟩`:`⟨*callout text*⟩`}` Draw a callout node containing ⟨*callout text*⟩ pointing to the glert node from ⟨*angle*⟩ (may be a direction keyword).

The default callout is rectangular and placed at a centre-to-centre distance 2.5 times the glert node "radius": i.e., the distance from centre to edge in the direction of ⟨*angle*⟩.

The ⟨*callout node options*⟩ can be used to change the callout appearance and placement. In particular, the `pos=`⟨*scale*⟩ option modifies the position scale relative to the glert node "radius", and `sep=`⟨*dimen*⟩ adds a further offset. If ⟨*angle*⟩ is a a keyword, `pos` moves along the corresponding direction defined by the glert node shape, whereas `sep` moves in the corresponding absolute direction (e.g. 45° for `north east`).

Options can be set for all callouts using the `glert callout options` style. Initially, this sets the defaults of `{rectangle callout, draw, pos=2.5, sep=0pt}`.

In the ⟨*color, ...*⟩ arguments, a colour name (as defined by the `xcolor` package) can generally be followed by other line or node options placed (along with the colour) in braces. Such options should take precedence over any default style, and over any options not grouped with the ⟨*color*⟩ argument. The other ways to set options are described in Section 7.3.7.

Multiple styles can be used together. It they set conflicting parameters (e.g. `box` vs. `ellipse`) then the last in the calling sequence will prevail.

7.3.4 Naming glerts

The name of the glert node can be set within the `\glert[*]` options with `node={name=`⟨*name*⟩`}`, or, more simply, `(`⟨*name*⟩`)`. The node can then be referenced by a `connect` glert, or from another Ti*k*Z picture (remembering to use `remember picture` and `overlay` as appropriate).

7.3.5 Glerts in math mode

Glert commands generally process math mode material transparently. In particular, there is no need to restore the math mode or style explicitly within the argument as there would be with most other boxed commands. However, TEX internals mean that to achieve this transparency, both the maths material and associated glert drawing must be rendered internally four times (in each of the four main maths styles), with only one rendering then being placed on the output page.

Although the added compilation induced by this quirk is generally acceptable, there is one context in which it creates a real difficulty. If a named glert is to be referenced (say, with the `connect` glert style), the same name is associated with all four nodes in turn, ending with the `\scriptscriptstyle` version. If the `\scriptscriptstyle` version isn't the one used in the final layout, then Ti*k*Z treats the final named node as though it were in the bottom left corner of the page.

To work around this issue `beamincr` makes it possible to specify the correct mathstyle explicitly, thus removing the need for the four versions. This is done by placing the style instruction as the *first* token in the ⟨*options*⟩ to the `\glert[*]` command. If a ⟨*glert style*⟩ or additional options are to be provided, these must follow. They can be separated from the mathstyle command by a comma, but this is not necessary.

Example:

```
\begin{equation*}% simple, but naming issues
\int_{\glert[(lo)]{-\infty}}^{\glert[(hi)]{+\infty}} e^{-i\omega t} d\omega = 2\pi\delta(t)
\end{equation*}

\begin{equation*}% contents rendered only once in the specified style
\int_{\glert[\scriptstyle (lo)]{-\infty}}^{\glert[\scriptstyle (hi)]{+\infty}}
       e^{-i\omega t} d\omega = 2\pi\delta(t)
\end{equation*}
```

Note that `align` environments also set their contents multiple times (to determine their sizes) and so glert naming doesn't work there either. There's no workaround for this at the moment.

7.3.6 Glerts in Ti*k*Z pictures

Glerts can be applied to nodes in Ti*k*Z pictures by setting the options below. In this use, no separate bounding box node is created. Thus, the extent of the node itself (including any glert-based modifications to the node) will ordinarily contribute to the size of the picture. However, any glert path additions will not, and so may extend outside the picture bounding box.

`tikz option:` `glert`={`<`⟨*overlay spec*⟩`>[`⟨*options*⟩`]`}

This option can be given to a Ti*k*Z node, to apply a glert specification according to ⟨*overlay spec*⟩. If no overlay specification is given, it is applied on all overlays. The ⟨*options*⟩ can specify glert styles or other options, as in the `\glert` command.

`tikz option:` `glert*`={`[`⟨*options*⟩`]`}

This option can be given to a Ti*k*Z node, to apply the glert specification when an enclosing `\glerts` command or action is active. The ⟨*options*⟩ can specify glert styles or other options, as in the `\glert*` command.

More than one `glert` and `glert*` option can be given to the same node, potentially applying different glerts on different overlays. Note, though, that glert specifications set by earlier `glert[*]`s will become the default for later ones.

7.3.7 Refining the glert appearance

In addition to the options defined explicitly by the standard glerts above, most other aspects of the glert appearance can be controlled (and extended). Stripped of various bits of internal munging, a glert essentially creates the following code:

```
\tikz[baseline,remember picture, glert picture ⟨state⟩]{
  \path[use as bounding box, glert bbox ⟨state⟩]
       node[anchor=base,inner sep=0pt,glert bbox ⟨state⟩]{\phantom{⟨contents⟩}};
  \pathnode[anchor=base, glert node ⟨state⟩]{⟨contents⟩};
  \path[⟨style definitions⟩, glert path ⟨state⟩] ⟨style path⟩;
}
```

where ⟨*state*⟩ is either `active` or `inactive`. (For the Ti*k*Zperts: the final path is only generated when necessary, and as a `\pgfextra` path created within the options to the preceding node. See Section 7.3.8.)

The `glert` ⟨*object*⟩ ⟨*state*⟩ styles can be used to modify the appearance of each of the paths (or set options for the entire picture). They can be set in two ways. The first is to provide the following option within `\useglert` or `\glert[*]` options, following any glert style declaration:

`glert option:` ⟨*object*⟩ ⟨*state*⟩=`{`⟨*options*⟩`}`

⟨*object*⟩=`picture` | `bbox` | `node` | `path`; ⟨*state*⟩=`active` | `inactive`

These options may be used within the argument to a `\useglert` or the options to `\glert` or `\glert*`. Each will append ⟨*options*⟩ to the `glert` ⟨*object*⟩ ⟨*state*⟩ style, thus modifying its appearance. If ⟨*state*⟩ is omitted it is taken to be `active`.

If called within `\useglert`, the changes will last until the next `\useglert` call, but will not apply to any `\glert[*]`s that set an explicit style. Options within `\glert[*]` are local to that instance.

(It is also possible to append to the `glert` ⟨*object*⟩ ⟨*state*⟩ style directly with `\tikzset` or similar. However, it is important not to overwrite the style entirely, unless you know exactly what you're doing!)

Longer lasting modifications can be achieved by altering glert-style-specific Ti*k*Z styles.

`tikz option: glert` ⟨*glert style*⟩ ⟨*object*⟩ ⟨*state*⟩`/.append style=`{⟨*options*⟩`}`

⟨*object*⟩=`picture` | `bbox` | `node` | `path`; ⟨*state*⟩=`active` | `inactive`

Modify the options of ⟨*object*⟩ in the active or inactive ⟨*state*⟩ of ⟨*glert style*⟩ glerts. These options can be set within `\tikzset`, `\useglert`, or `\glert[*]`. In the first two cases, the effect will apply throughout the current TEX block. In the latter, it applies just to the glert in question (and so is effectively just a more verbose version of the option above).

Example:

```
\tikzset{glert box node active/.append style={inner sep=4pt}}
```

adds a little extra space between the box and text in all subsequent `\glert[box]` glerts.

Any ⟨*further options*⟩ provided to `\useglert` or `\glert[*]` that aren't specific to glert handling are passed to Ti*k*Z for interpretation. Thus, the extra space above could be added within a `\glert` more simply.

Example:

```
\glert{box, inner sep=4pt}{text}
```

In this case, the `inner sep=4pt` option would apply to all nodes within the picture: however the `bbox` node sets its own value to 0pt explicitly and so will be unaffected. To also increase the space reserved for the glert, use

Example:

```
\glert{box, inner sep=4pt, bbox={inner sep=4pt}}{text}
```

7.3.8 Defining new glerts

`\makeglertstyle{`⟨*glert style*⟩`}`

Create and initialise a new glert style called ⟨*glert style*⟩. This creates the appropriate Ti*k*Z styles and sets up the code for these to be installed by `\useglert{`⟨*glert style*⟩`}` or `\glert[`⟨*glert style*⟩`]` and similar. The initial styles are all empty, so the glert is equivalent to `null`.

`\defineglertstyle{`⟨*glert style*⟩`}{`⟨*object*⟩`}{`⟨*state*⟩`}[`⟨*default setting*⟩`]{`⟨*tikz style*⟩`}`

Define the ⟨*tikz style*⟩ assigned to ⟨*object*⟩ when a ⟨*glert style*⟩ glert is in ⟨*state*⟩. As usual, the ⟨*tikz style*⟩ may accept an argument (or value) which can be accessed in the style definition as `#1`. The ⟨*default setting*⟩ provides an optional default for this value.

`\defineglertpath{`⟨*glert style*⟩`}{`⟨*state*⟩`}{`⟨*tikz path commands*⟩`}`

Add the following option to the ⟨*glert style*⟩ `node` ⟨*state*⟩ style:

`append after command={\pgfextra\path[glert path` ⟨*state*⟩`]` ⟨*tikz path commands*⟩`;\endpgfextra}`

This causes the path described by ⟨*tikz path commands*⟩ to be created after the node, with options defined by `glert path` ⟨*state*⟩. Any action on the path (such as `draw`, `fill`, `shade` ...) should be specified in the options if needed. The following commands are available to help create the path (in addition to Ti*k*Z's `\tikzlastnode`):

`\glertnode` refers to the glert node

`\glertbbox` refers to the glert bbox node

`tikz option: glert angle parser={[`⟨*options*⟩`]`⟨*angle*⟩`:`⟨*target*⟩`}`

This option is designed to accept a label-type argument to a glert and parse it into components that can be used by later options or glert paths. It also installs two options for local use within ⟨*options*⟩: `pos=`⟨*pos*⟩ and `sep=`⟨*sep*⟩.

The effect is such that after parsing and evaluation of ⟨*options*⟩ (for `pos` and `sep`) the following Ti*k*Z options and commands are available :

```
glert angle options/.style=⟨options⟩
glert angle=⟨angle⟩
glert angle target=⟨target⟩
glert angle pos=⟨pos⟩
glert angle sep=⟨sep⟩
\def\glertangleoptions{⟨options⟩}
\def\glertangle{⟨angle⟩}
\def\glertangletarget{⟨target⟩}
\def\glertanglepos{⟨pos⟩}
\def\glertanglesep{⟨sep⟩}
\def\glertangleplace{($(\glertnode.center)!\glertanglepos!(\glertnode.\glertangle)
                        + (\glertangle:\glertanglesep)$)}
```

The final command defines a coordinate (⟨*pos*⟩ * `\glertnode` "radius" + ⟨*sep*⟩) away from the centre of `\glertnode` in the direction ⟨*angle*⟩.

Here are the definitions of some builtin glert styles:

```
\makeglertstyle{alert}
  \defineglertstyle{alert}{node}{active}[red]{text=#1}
\makeglertstyle{box}
  \defineglertstyle{box}{node}{active}[red]{very thick, draw=#1}
\makeglertstyle{ellipse}
  \defineglertstyle{ellipse}{node}{inactive}{ellipse}
  \defineglertstyle{ellipse}{node}{active}[red]{ellipse, very thick, draw=#1}
\makeglertstyle{highlight}
```

```
  \defineglertstyle{highlight}{node}{active}[red]{fill opacity=0.2, text opacity=1, fill=#1}
\makeglertstyle{uline}
  \defineglertpath{uline}{active}{(\glertnode.south east) -- (\glertnode.south west)}
  \defineglertstyle{uline}{path}{active}{draw,red,very thick,#1}
\makeglertstyle{spotlight}
  \defineglertstyle{spotlight}{node}{active}[red!50]{ellipse,path fading=spotlight,fill=#1}
  \defineglertstyle{spotlight}{node}{inactive}{ellipse}
\makeglertstyle{connect}
  \defineglertpath{connect}{active}{(\glertnode) to #1}
  \defineglertstyle{connect}{path}{active}{draw,red,very thick,<->}
\makeglertstyle{callout}
  \tikzset{glert callout options/.style={rectangle callout, draw, pos=2.5, sep=0pt}}
  \defineglertpath{callout}{active}{
    node[glert angle parser={#1}, glert callout options, glert angle options, at=\glertangleplace,
         callout absolute pointer=(\glertnode.\glertangle), node contents={\glertangletarget}]}
```

Wherever appropriate, the argument `#1` is applied at the end of the options list. This ensures that any options passed directly to the glert style will override the defaults that come before. Glerts that use a shaped node also apply the shape when inactive, so as to avoid "bouncing" caused by bounding box changes when used within a Ti*k*Z picture (when there is no separate bbox node; Section 7.3.6).

8 Selectively repeating frames

These functions control the (re)display of frames.

`\againframe<`⟨*overlay spec*⟩`>[<`⟨*default spec*⟩`>][`⟨*options*⟩`]{`⟨*frame label*⟩`}`

This is a BEAMER command to repeat a labelled frame, allowing the ⟨*overlay spec*⟩ and other options to be modified. It is modified in `beamincr` so that the ⟨*overlay spec*⟩ respects `beamincr` increment labels, and so that any `\framescanonly` commands in the orginal frame contents are ignored (thus allowing the frame contents to be rendered).

`\handoutframe[`⟨*modes*⟩`]<`⟨*overlay spec*⟩`>[<`⟨*default spec*⟩`>][`⟨*options*⟩`]{`⟨*frame label*⟩`}`

When compiled in one of ⟨*modes*⟩ (defaults to `handout`, but can include more than one, e.g., `[beamer|handout]`), render the specified overlays from the named frame. If ⟨*overlay spec*⟩ is omitted, all the overlays are rendered. No output is produced in modes other than ⟨*modes*⟩.

The code works by switching temporarily to `beamer` mode as that seems to be the only way to produce more than one overlay per frame in `handout`, `trans` and `article` modes, although this means it may behave poorly with any mode-specific material within the frame. The idea is from `https://tex.stackexchange.com/questions/455444/beamer-overlays-and-handout-exclude-frames-from-handout/455459#455459`.

Unfortunately, the natural code

Example:

```
\begin{frame}<handout:0>[label=twolists]
  ...
\end{frame}
\handoutframe<1,/halfway/,/done/>{twolists}
```

fails, because the `<handout:0>` spec stops the increment labels from being defined. If running a recent LaTeX compiler (post 2021) the command `\framescanonly<handout>` described below provides a workaround.

Example:

```
\begin{frame}[label=twolists]
  \framescanonly<handout|trans>%
  ...
\end{frame}
\handoutframe[handout|trans]<1,/halfway/,/done/>{twolists}
```

`\framescanonly<`⟨*modes*⟩`>`

Scan the current frame without producing any output. This is similar to a `<mode:0>` specification to `\begin{frame}`, but as the frame is still scanned it allows side effects such as increment label definitions. The `\framescanonly` command should be placed inside the frame contents. It is only available in recent versions of LaTeX with extended hook support; a warning is printed in other cases. If used in `beamer` mode the frame will be reprocessed for every overlay. This behaviour can be avoided by also including a `<beamer:1>` or `<beamer:-1>` (but not `<beamer:0>`!) or equivalent specification to `\begin{frame}`. although it may be useful to fully expand advanced increment references when increments are reset (see Section 4.4).

The command has no effect when the frame is recalled with `\againframe` or `\handoutframe`, allowing both commands to render the frame contents. If it is necessary to suppress the output of (say) `\againframe` in certain modes, this can be achieved with the usual `<`⟨*mode*⟩`:0>` specification.

An example use appears above.

`\allowframescanonly[`⟨*flag*⟩`]`

Disable (with ⟨*flag*⟩ = 0) or enable (with ⟨*flag*⟩ > 0 or omitted) the effect of `\framescanonly`.

9 Internals

These sections discuss more background and some implementation details. This is only likely to be of interest to users who wish to extend the approach.

9.1 Pauses, increments and the `beamerpauses` counter

Both `\pause` and incremental overlay specifications access the same underlying counter called `beamerpauses`, but they use them in different ways.

`\pause`

increments `beamerpauses` and then sets subsequent material on the slide given by the incremented `\value{beamerpauses}`.

`\onslide<+->`

increments `beamerpauses` but then sets subsequent (or argument) material on the slide corresponding to the *previous* value of `beamerpauses`.

`\onslide<.->`

leaves `beamerpauses` alone, but sets subsequent (or argument) material on the slide given by `\value{beamerpauses}-1`, unless `\value{beamerpauses}=1` in which case it puts subsequent material on slide 1.

This conflict in interpretation of the `beamerpauses` counter can cause unintuitive effects. The incremental specfication model is far more flexible and powerful, and so the commands of this package can all be interpreted in terms of an *increment number* which ordinarily equals `max(\value{beamerpauses}-1,1)`. In fact, internally they all use the `beamerpauses` counter with this offset. Thus, when commands like `\resetincr` set the increment value, they set `beamerpauses` to the increment + 1. This value then behaves sensibly with `<+>` etc. specifications, but not with `\pause`.

An exception is when the current increment is set to 0, either explicitly or by using an advanced (or otherwise undefined) increment reference. In this case `beamerpauses` is also set to 0, not 1. This is because `beamincr` references use the 0 value to detect the undefined reference, and so suppress offsets and ranges as described in Section 4.4. However, subsequent uses of BEAMER's `<+>` specification will increment `beamerpauses`, potentially placing text on earlier slides than intended. This behaviour can be avoided (at the expense of more typing) by using `<!/.(1)/>` instead.

9.2 Overlay specification parsing routines

The `beamincr` overlay and action specification extensions work by injecting various parsing routines into the core BEAMER parser (called `\beamer@masterdecode`), before calling the original function. These parsers are also available as user commands, and may be helpful for debugging (though see also `\beamincrdebug` below).

`\parseincludedefaultspec{`⟨*overlay spec*⟩`}`

Replace any <~>fields in ⟨*overlay spec*⟩ with the current default specification.

`\parseincrspec{`⟨*overlay spec*⟩`}`

Interpret any text enclosed in /s within ⟨*overlay spec*⟩ as an increment specification, replacing each with the corresponding numerical values (including offset). Also processes any open ranges internal to the label as described in Section 4.4.

`\parseresetspec{`⟨*overlay spec*⟩`}`

`\parselabelspec{`⟨*overlay spec*⟩`}`

9.3 Interface with beamer internals

Many `beamincr` extensions work to pre-process material before passing it to standard BEAMER or other commands. This section details the few cases where it proved necessary to interface directly with BEAMER internals.

As described above, increment control is achieved by reading or setting the `beamerpauses` counter. The user-visible increment number corresponds to `max(\value{beamerpauses}-1,1)`, so as to consistently match the slide numbers on which increment-assigned material appears. Many commands also read and modify BEAMER internal macros used as variables, notably `\beamer@defaultospec` (the current default overlay specification) and `\beamer@againname` (the label of the current frame, used to identify it in calls to `\againframe`). In particular, `\beamer@againname` is incorporated into the internal name associated with `beamincr` labels.

Extensions to the overlay and action specification syntax require modification to the BEAMER parsing routines, to inject the parsing routines described in Section 9.2. This is achieved using the following code.

```
\let\beamer@masterdecode@orig=\beamer@masterdecode
\def\beamer@masterdecode#1{%
  \edef\parsed@spec{\parseincludedefaultspec{#1}}%
  \edef\parsed@spec{\x@\parseincrspec\x@{\parsed@spec}}%
  \edef\parsed@spec{\x@\parseresetspec\x@{\parsed@spec}}%
  \edef\parsed@spec{\x@\parselabelspec\x@{\parsed@spec}}%
  \debug@message{masterdecode: <\unexpanded{#1}> -> <\parsed@spec>^^J}%
  \x@\beamer@masterdecode@orig\x@{\parsed@spec}%
}
```

where `\x@` is an internal abbreviation for `\expandafter`.

The command `\againframe` must be modified separately, both to interpret increment references and to inhibit any `\framescanonly` in the contents. BEAMER uses a cascade of internal commands to read the various possible optional arguments. These are retained, with the change happening at the inner-most command.

```
\let\beamer@@@againframe@orig=\beamer@@@againframe
\def\beamer@@@againframe<#1>[#2][#3]#4{%
  \edef\@scanstate{\@ifallowscanonlystate}%
  \allowframescanonly0%
  \beamer@@@againframe@orig<\parseincrspec{#1}>[#2][#3]{#4}%
  \x@\allowframescanonly\@scanstate%
}
```

The `\handoutframe` command calls this redefined `\againframe` internal after resetting the current mode to `beamer`.

9.4 Debugging

`\beamincrdebug{`⟨*flag*⟩`}`

Turn on (⟨*flag*⟩ > 0) or off (⟨*flag*⟩ = 0) debugging mode. When on, compilation generates messages in the terminal output and log file describing the rewriting actions that `beamincr` performs.

10 Reference

10.1 Increment, overlay and action specifications

⟨*num*⟩	integer	explicit reference to slide number
⟨*label*⟩	text	value assigned by `\incrlabel{`⟨*label*⟩`}` or `<=>`
⟨*incrref*⟩	⟨*label*⟩`(`⟨*offset*⟩`)`	= `value{`⟨*label*⟩`}+`⟨*offset*⟩
|	`.(`⟨*offset*⟩`)`	= ⟨*current increment*⟩`+`⟨*offset*⟩
|	`^(`⟨*offset*⟩`)`	= ⟨*first overlay number*⟩`+`⟨*offset*⟩
|	`$(`⟨*offset*⟩`)`	= ⟨*final overlay number*⟩`+`⟨*offset*⟩
|	`$*(`⟨*offset*⟩`)`	= ⟨*highest overlay used so far*⟩`+`⟨*offset*⟩
|	`@(`⟨*offset*⟩`)`	= ⟨*most recently used overlay*⟩`+`⟨*offset*⟩
|	`!(`⟨*offset*⟩`)`	= ⟨*last increment reset*⟩`+`⟨*offset*⟩
⟨*incrnum*⟩	⟨*num*⟩ | `/`⟨*incrref*⟩`/`	increment number for `\resetincr`, `\fromincr`, `\incrlabel`
⟨*incr*⟩	`/`⟨*incrref*⟩`/`	= `value{`⟨*incrref*⟩`}` (including ⟨*offset*⟩)
|	`.(`⟨*offset*⟩`)`	= ⟨*current increment*⟩`+`⟨*offset*⟩
|	`+(`⟨*offset*⟩`)`	= `++`⟨*current increment*⟩`+`⟨*offset*⟩
⟨*slide*⟩	⟨*num*⟩ | ⟨*incr*⟩	
⟨*range*⟩	⟨*slide*⟩ | ⟨*slide*⟩`-`⟨*slide*⟩	(at least one ⟨*slide*⟩ must be present)
|	`/`⟨*incrref*⟩`-/`	= `0-0` if ⟨*incrref*⟩ evaluates to 0 or is undefined and `\allowundefinedincrlabels` is true
⟨*mode*⟩	BEAMER mode	`beamer` | `trans` | `handout` | `presentation` | `article` | `all`
⟨*modes*⟩	⟨*mode*⟩$_1$`|`⟨*mode*⟩$_2$`|...`	
`<`⟨*overlay spec*⟩`>`		
`<*>`		active on all slides
`<`⟨*range*⟩$_1$`,`⟨*range*⟩$_2$`,...>`		active for slides within ⟨*range*⟩s
`<~>`		copy default specification
`<`⟨*mode*⟩`:`⟨*overlay spec*⟩`>`		only apply ⟨*overlay spec*⟩ in ⟨*mode*⟩
`<`⟨*overlay spec*⟩$_1$`|`⟨*overlay spec*⟩$_2$`|...>`		apply different ⟨*overlay spec*⟩s in different modes
`<`⟨*action spec*⟩`>`		
`<`⟨*overlay spec*⟩`>`		
`<resetincr@`⟨*slide*⟩`>`		set ⟨*current increment*⟩ to ⟨*slide*⟩
`<!`⟨*incrnum*⟩`-`⟨*incrnum*⟩`>`		= `<resetincr@`⟨*incrnum*⟩`|`⟨*incrnum*⟩`-`⟨*incrnum*⟩`>`
`<=(`⟨*offset*⟩`)>.../(`⟨*offset*⟩`)`⟨*label*⟩`/ a`		set ⟨*label*⟩ to increment in effect after ⟨*action spec*⟩ +⟨*offset*⟩s
`<defaultspec@`⟨*range*⟩`>`		set default spec within argument to ⟨*range*⟩ evaluated in ⟨*action spec*⟩
`<~`⟨*range*⟩`>`		= `<`⟨*range*⟩`|defaultspec@`⟨*range*⟩`>`
`<eqnum@`⟨*range*⟩`>`		insert (and advance) `\theequation` tag for slides in ⟨*range*⟩
`<alert@`⟨*range*⟩`>`		`\alert` argument for slides in ⟨*range*⟩
`<alerts@`⟨*range*⟩`>`		activate `\alert*` commands in argument in ⟨*range*⟩
`<point@`⟨*range*⟩`>`		`\point` and activate `\point*` commands in ⟨*range*⟩
`<pointers@`⟨*range*⟩`>`		activate `\point`$\widehat{*}$ commands in argument in ⟨*range*⟩
`<`⟨*mode*⟩`:`⟨*action spec*⟩`>`		only apply ⟨*action spec*⟩ in ⟨*mode*⟩
`<`⟨*action spec*⟩$_1$`|`⟨*action spec*⟩$_2$`|...>`		apply all ⟨*action spec*⟩s subject to any ⟨*mode*⟩ restrictions

10.2 List of commands and environments

Increment labels and references

Incremental environments

Alignment environments

Directing attention

Selectively repeating frames

Beamer pause commands

Internal parsing commands

Debugging