Enhanced BEAMER increments: the `beamincr` package

Maneesh Sahani

October 22, 2023

The `beamincr` package provides a set of extensions and enhancement to 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 in-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 7. 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[`⟨*incr*⟩`]`

This command resets the increment number to 1, or to the value defined by ⟨*incr*⟩ if given. It doesn't directly affect the slide on which subsequent text appears, but it does change effect of subsequent `<+>` or `<.>` increments. The 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}
```

The argument ⟨*incr*⟩ must either be a number or be an increment specification 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<`⟨*incr*⟩`>`

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 `\afterframe` or `\handoutframe` (described below).

`\incrlabel<`⟨*incr*⟩`>{`⟨*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 any of the characters `.!=` or contain any of `()-~`.

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.

`\incrref{`⟨*incr spec*⟩`}`

This command returns the increment number defined by increment specification ⟨*incr spec*⟩ as described below.

The general form of an increment specification is

`increment specification`: `label(offset)`

The label can be a string assigned by a call to `\incrlabel`, or the special label '`.`' which refers to the current increment (this is subtly different to the incremental overlay specification '`.`'). A further special label '`!`' is introduced in Section 3.1. The ⟨*offset*⟩, if given, is added to the increment indicated by the label. It can be negative.

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

An example:

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 increment specification (i.e. `label(offset)`), while `\resetincr`, `\fromincr` and `\incrlabel` expect a restricted overlay specification that might be an increment specification in `//` (e.g. `/label(offset)/`) or just a number. Standard overlay-aware commands should all accept overlay specifications that include increment specs.

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, by default, just show a single slide with all the overlays collapsed. See Section 6.

3.1 Setting increments in overlay action specifications

It is possible to use the BEAMER syntax for actions in overlays to reset the increment number. This can be done using the explicit `resetincr@`⟨*increment*⟩ action or with an implicit `<!`⟨*increment*⟩`>` specification.

Example:

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

The increment number can be replaced by 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.

Example:

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

It is possible to issue multiple `!` commands in one overlay spec but only the first of them will take effect.

According to the user guide, BEAMER actions are only supported by overlay specifications to `\action`, `\item`, the `actionenv` environment and block environments like `block` and `theorem`. This package adds the fields of incremental (Section 4) and incremental alignment environments (Section 5) to this list. In the absence of any action specifications, `\action` acts like `\uncover` (or `\onslide` with an argument). Note that `\fromslide` (Section 2) can act like an `\onslide` declaration while also setting the increment.

3.2 Assigning labels from overlay action specifications

The `<...|=(`⟨*offset*⟩`)| ...>` 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]
\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}
```

the first two action calls set the label `x` to 5. The latter two 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 `<`⟨*incr*⟩`>` argument to `\incrlabel`.

If no `/label/` is found or if the name is empty, the action tries to do nothing quietly. This makes it possible to use an `=` in a default spec, but only assign a label to some of the 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 an explicit `//` pair appears first. You have been warned!

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

3.3 Extending default overlay or action specifications

By default, 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.

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 5), a `~` will incorporate the field-specific default.

3.4 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 6).

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}
```

4 Incremental environments

The `beamincr` package provides a set of 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 5

`\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 tikz 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 treated just like ⟨*next contents*⟩: it is executed with the action specification given by `<`⟨*pre-next specification*⟩`>` if present, or else the default specification. 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. In particular, this avoids side effects from, e.g., `<+>` specifications in the default. (This does not apply to later empty ⟨*next contents*⟩.)

A counter called `next`, is set to 0 in the pre-next field and then advanced at every `\next`.

Example:

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

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

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

```
\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.

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 the arguments, but is otherwise the same as \begin{incrementaldodef} ...

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

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

4.1 TikZ-based incremental environments

The following environments use TikZ picture commands to create their effects. They will only be defined if TikZ is loaded before beamincr.

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

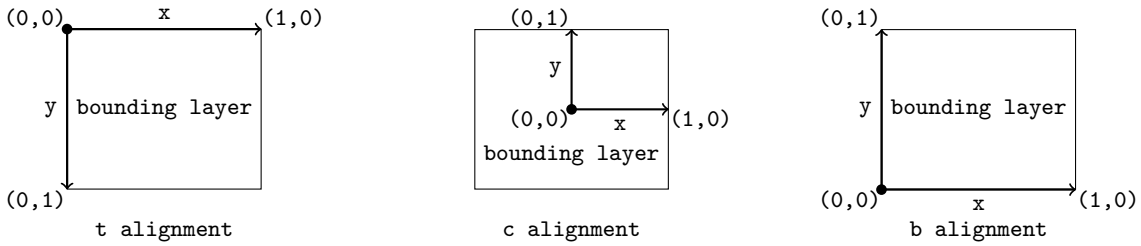
Set ⟨*next contents*⟩ within overlayed TikZ nodes. This may have the effect of later text appearing to be layered on top of earlier material. If the specifications place material on mutually exclusive increments (e.g. with <+>) then this has a similar effect to BEAMER's overprint environment. In particular, 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 TikZ options provided to the nodes, in one of three ways: by setting options in the incremental layer TikZ 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 passed directly to TikZ. The default origin of the 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 tikz option `fit layers` can be given to a node to surround all the layers defined so far in the `incrementallayers` environment (see the tikz fit library).

There is also a new environment called `boundinglayerscope` which creates a TikZ `scope` with the following additions. A node called `(bounding layer)` is created to (tightly) fit the layers defined so far, and the `xy` coordinate system in the scope is redefined according to the alignment of the most recent layer, or a `tcb` alignment option to `boundinglayerscope`. 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).



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

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

5 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. However, 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.

The following four environments are equivalent, apart from equation numbering:

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

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

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

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

Each pre-processes the input to `align[*]` to place an `\action<>{}` command around each field, defined as the material appearing between successive `&` and/or `\\` 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, these sequence is repeated. The default specification values can be changed by calling `\setincrementalenvspec{align*}{<new defaults>}` 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 or `=` label assignments.

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 using either 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 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<>}

% 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<>}
```

It may be wise to put the `\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 environments. These can be avoided by using a form like `\nonumber\\[-3ex]` instead.

At present, equation numbers aren't overlay aware.

The incremental forms `incrementalgather`, `gather<>`, `incrementalgather*` and `gather*<>` are also created when `beamincr` is loaded. Although these contain only one field per line, automatic access to BEAMER and `beamincr` actions as these lines are processed can be useful.

`\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 fields ae uncovered one-by-one rather than a line at a time, 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 with 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.

`\CreateIncrementalAlignmentEnvironment`{⟨*base*⟩}[⟨*Nopts*⟩]{⟨*Nreqs*⟩}[<default specification>]

Create an increment-aware version of an alignment environment. The ⟨*base*⟩ 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`⟨*base*⟩ or ⟨*base*⟩`<>`.

`\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*}
```

`\usenonincrementalenv`{⟨*name*⟩}

Restore the normal non-incremental behaviour of subsequent `name` environments.

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

Set the default specification for incremental environments of type `name`.

6 Misc helper functions

These functions may prove useful under some circumstances.

`\handoutframe`[⟨*mode spec*⟩]<⟨*overlay specification*⟩>{⟨*frame label*⟩}

This command only produces output when compiled in ⟨*mode spec*⟩ (which defaults to `handout`, but can include more than one, e.g., `[beamer|handout]`), in which case it renders the specified overlays from the named frame. If ⟨*overlay specification*⟩ is omitted, all the overlays are rendered. 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<`⟨*mode specification*⟩`>`

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 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 3.4).

An example use appears above.

`\allowframescanonly[`⟨*flag*⟩`]`

This commands disables (with ⟨*flag*⟩ = 0) or enables (with ⟨*flag*⟩ > 0 or omitted) the effect of `\framescanonly`.

`\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.

7 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.

7.1 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 3.4. 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.

7.2 Overlay specification parsing routines

The `beamincr` overlay 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` above).

`\parsedefincspec{`⟨*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 3.4.

`\parseresetspec{`⟨*overlay spec*⟩`}`

`\parselabelspec{`⟨*overlay spec*⟩`}`