The typed-checklist package*

Richard Grewe
r-g+tex@posteo.net

2022-05-29

Abstract

The main goal of the typed-checklist package is to provide means for typesetting checklists in a way that stipulates users to explicitly distinguish checklists for goals, for tasks, for artifacts, and for milestones – i.e., the *type* of checklist entries. The intention behind this is that a user of the package is coerced to think about what kind of entries he/she adds to the checklist. This shall yield a clearer result and, in the long run, help with training to distinguish entries of different types.

1 Motivation and Disambiguation

The development of this package was driven with two goals in mind:

1. having a package with which one can easily typeset checklists and in a way that separates content from layout;
2. having a thinking tool that helps distinguishing between goals and tasks.

The first goal felt natural to me since from time to time I manage checklists in LaTeX documents, mostly because I like it when the result looks typeset nicely. The second goal arose from an observation about some of my own checklists as well as checklists created by others: Quite frequently, the checklists mixed goals and tasks or had goals formulated as tasks and vice versa. As a consequence, the checklists were formulated unnecessarily unclear and were more difficult to understand by others.

This package approaches particularly the second goal by providing checklists with a *type*. A checklist of a particular type shall then only contain entries of this type.

While the package allows one to define custom checklist types (see Section 4), it comes with four basic types: `Artifact`, `Goal`, `Milestone`, and `Task`. In this documentation, the terms "artifact", "goal", "milestone", and "task" will be used along the lines of the following definitions (highlights added):

*This document corresponds to typed-checklist v2.1, dated 2022/05/28. The package is available online at http://www.ctan.org/pkg/typed-checklist and https://github.com/Ri-Ga/typed-checklist.

artifact:
- "An object made or shaped by human hand." (Wiktionary)

goal:
- "An observable and measurable end result having one or more objectives to be achieved within a more or less fixed timeframe." (BusinessDictionary.com)
- "the end toward which effort is directed" (Merriam-Webster)
- "The object of a person's ambition or effort; an aim or desired result" (Oxford Dictionaries)
- "A result that one is attempting to achieve." (Wiktionary)

milestone:
- "An important event [...] in the life of some project" (Wiktionary)

task:
- "a usually assigned piece of work often to be finished within a certain time" (Merriam-Webster)
- "A piece of work done as part of one's duties." (Wiktionary)

We could connect the four terms as follows. Typically, the "piece of work" that constitutes a task is performed for achieving some goal. One can also say that a goal serves as a reference point for why and how one should perform certain tasks. A goal can be that a particular artifact or set of artifacts is available at some point in time. A milestone is a group of goals whose achievement is of importance for something bigger. These connections suggest that nesting different types of checklists is reasonable – and it is supported by the typed-checklist package.

2 Recommendations for Structuring Checklists

The typed-checklist package allows checklists of different types as well as of identical types to be nested. That is, within a checklist, another checklist can be placed. The following list discusses some combinations of nested checklist types and provides some recommendations of what types could be nested for particular purposes and what types should better not be nested.

1. tasks in goals . . . . . ✓

   This nesting combination could be used for listing tasks whose accomplishment would lead to the satisfaction of the superordinated goal.

2. goals in goals . . . . . ✓

   This nesting combination could be used for explicitly listing sub-goals (and sub-sub-goals and . . .) to a goal. That is, using this nesting combination you can express the result of breaking down goals into sub-goals. Used reasonably, this nesting should be used in a way that the sub-goals, when achieved, yield the superordinated goal to be achieved (at least with high probability and/or to a significant extent).

3. tasks in tasks . . . . . ✓

   This nesting combination could be used for listing all sub-tasks to a task. That is, using this nesting combination you can express the result of breaking down tasks into sub-tasks.

4. goals in milestones . . . . . ✓

   This nesting combination could be used for listing all goals that must be achieved, at a particular date, for calling a milestone achieved.

5. artifacts in milestones . . . . . ✓

   This nesting combination could be used for listing all artifacts that must exist, at a particular date, for calling a milestone achieved.

6. goals in tasks . . . . . ☠

   This nesting lacks a clearly recognizable meaning. The use of this kind of nesting might be an indicator for a misunderstanding of goals or tasks, or it might be the result of too vague formulations of goals or tasks that do not reveal that something is wrong in the planning.

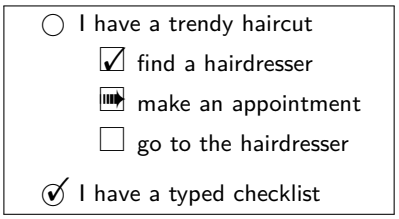
7. milestones in milestones . . . . . ☠

   A milestone, as cited, is an important event. Having sub-milestones would blur the notion of important events by introducing multiple levels of important events. Instead of nesting milestones, one could nest goals or artifacts in milestones to express intermediate stages of a milestone.

3 Basic Usage

The following example demonstrates a basic use of the package.

```
\documentclass{article}
\usepackage{typed-checklist}
\begin{document}
\begin{CheckList}{Goal}
  \Goal{open}{I have a trendy haircut}
    \begin{CheckList}{Task}
      \Task{done}{find a hairdresser}
      \Task{started}{make an appointment}
      \Task{open}{go to the hairdresser}
    \end{CheckList}
  \Goal{achieved}{I have a typed checklist}
\end{CheckList}
\end{document}
```



The example contains a checklist for goals and the first goal contains a checklist for tasks. Checklist entries have a status and a description. In the typeset result, the checklist type is reflected by a basic symbol (an empty circle for a goal and an empty box for a task) that is decorated depending on the status (e.g., with a check mark). The entry's description is shown next to the symbol.

3.1 Checklists

`\begin{CheckList}[`⟨*options*⟩`]{`⟨*type*⟩`}`
`\end{CheckList}`

Checklists are created via the `CheckList` environment. The ⟨*type*⟩ parameter determines the type of all checklist entries in the environment. The typed-checklist

package comes with four predefined types: `Goal`, `Task`, `Artifact`, and `Milestone`. Each of the types comes with a macro of the same name as the type. With this macro, the entries of the checklist can be created.

The ⟨*options*⟩ can be a comma-separated list of ⟨*key*⟩=⟨*value*⟩ pairs. Table 1 on page 8 shows the keys and possible values that can be set.

Defaults for checklist options can also be specified globally, either through package options or through the `\CheckListSet` macro.

`\CheckListSet{`⟨*options-list*⟩`}`

This macro takes a comma-separated ⟨*options*⟩ list and sets these options for all subsequent checklists.

A checklist can be viewed as a list of entries (even if the layout is actually tabular). The macros for creating the entries are described next.

3.2 Checklist Entries

`\Goal[`⟨*options*⟩`]{`⟨*status*⟩`}{`⟨*description*⟩`}`

Inside a checklist of type `Goal`, the `\Goal` macro specifies a goal. Every goal comes at least with a ⟨*description*⟩ and a ⟨*status*⟩. The ⟨*description*⟩ can, technically, be anything that is displayable in the given checklist `layout`. However, for the purpose of a meaningful checklist, the ⟨*description*⟩ should be a clear description of a goal in a full sentence[1]. The ⟨*status*⟩ parameter selects the most recent known status of the goal. This parameter can assume any of the following values[2]:

`achieved`	This value specifies that the goal has been achieved. Depending on how the ⟨*description*⟩ was formulated, this might mean that in the respective situation the ⟨*description*⟩ is a true statement.
`dropped`	This value specifies that the goal was a goal once but is no longer a goal that shall be pursued. This value allows one to preserve historical information about a checklist.
`unclear`	This value specifies that the goal somehow exists but is not yet clear enough to those who pursue the goal (or: who typeset the checklist) for actually pursuing the goal.
`open`	This value specifies the negation of all aforementioned values. That is, the goal is clear but neither achieved yet nor dropped.

The ⟨*options*⟩ allow one to specify further details about the goal. The ⟨*options*⟩ must be a possibly empty, comma-separated list of ⟨*key*⟩=⟨*value*⟩ pairs. The ⟨*key*⟩ must be one of the following values[3]:

`who`	This option declares who is responsible for making sure the checklist entry is addressed. Remember to put the value in curly braces if it contains commas.

[1] Incomplete sentences tend to be less clear.
[2] See Section 4.1 to find out how custom states can be defined
[3] See Section 4.2 to find out how custom ⟨*key*⟩s can be defined.

`deadline`	This option declares a deadline for the checklist entry, i.e., a date until which the entry must be addressed at latest. The format for specifying deadlines is determined by the checklist options `input-dates` and `strict-dates`.
`label`	This option declares a label name for the checklist entry. This is analogous to the `\label` macro of LaTeX. The entry's label is displayed next to the entry. A reference to a labeled checklist entry can be made using the `\ref` macro of LaTeX.

`\Task[`⟨*options*⟩`]{`⟨*status*⟩`}{`⟨*description*⟩`}`

Inside a checklist of type `Task`, the `\Task` macro specifies a task. Every task comes at least with a ⟨*description*⟩ and a ⟨*status*⟩. The ⟨*description*⟩ can, technically, be anything that is displayable in the given checklist `layout`. However, for the purpose of a meaningful checklist, the ⟨*description*⟩ should be a clear description of a task in a full sentence, possibly in imperative form. The ⟨*options*⟩ parameter can be set as documented for the `\Goal` macro on page 5. The ⟨*status*⟩ parameter selects the most recent known status of the task. This parameter can assume any of the following values:

`open`	This value specifies that the task is still planned but has not yet been started.
`dropped`	This value specifies that the task was originally planned but is no longer part of the plan.
`unclear`	This value specifies that the task itself or its current status is unclear.
`started`	This value specifies that someone has started to perform the task, but has not finished yet.
`done`	This value specifies that someone has accomplished the task. Depending on the clarity and level of detail of the ⟨*description*⟩, whether accomplishing the task yielded a meaningful outcome might be more or less subjective to the person who accomplished the task.

`\Artifact[`⟨*options*⟩`]{`⟨*status*⟩`}{`⟨*description*⟩`}`

Inside a checklist of type `Artifact`, the `\Artifact` macro specifies an artifact. Every artifact comes at least with a ⟨*description*⟩ and a ⟨*status*⟩. The ⟨*description*⟩ can, technically, be anything that is displayable in the given checklist `layout`. However, for the purpose of a meaningful checklist, the ⟨*description*⟩ should be a clear identification of the artifact and its required attributes. The ⟨*status*⟩ parameter selects the most recent known status of the artifact. This parameter can assume any of the following values:

`missing`	This value specifies that the artifact is missing yet.
`dropped`	This value specifies that the artifact was originally planned but is no longer part of the plan.

`unclear`	This value specifies that the artifact itself or its current status is unclear.
`incomplete`	This value specifies that some non-negligible parts of the artifact exist but the artifact does not yet exist in its final form.
`available`	This value specifies that the artifact exists and available.

`\Milestone[`⟨*options*⟩`]{`⟨*status*⟩`}{`⟨*description*⟩`}`

Inside a checklist of type `Milestone`, the `\Milestone` macro specifies a milestone. Every milestone comes at least with a ⟨*description*⟩ and a ⟨*status*⟩. The ⟨*description*⟩ can, technically, be anything that is displayable in the given checklist `layout`. However, for the purpose of a meaningful checklist, the ⟨*description*⟩ should be a clear identification of what has to exist or must have been fulfilled. The ⟨*status*⟩ parameter selects the most recent known status of the milestone. This parameter can assume any of the following values:

`open`	This value specifies that the milestone has not yet been achieved.
`achieved`	This value specifies that the milestone has been achieved.

3.3 Comprehensive Example

The example in Listing 1 on page 9 shows the use of nested checklists and the use of various checklist and entry options. The example deliberately mixes different date formats for the sake of demonstration, but this should normally be avoided as it reduces legibility.

4 Customized Checklists

The typed-checklist package comes with a set of layouts, checklist types, checklist entry states, and checklist entry options. These together shall provide everything needed for typesetting even checklists with complex structures. When the default is not enough, you can use the macros described in this section for creating your own layouts, types, states, and options.

4.1 Defining Checklist Types and Entry States

`\CheckListAddType{`⟨*type*⟩`}{`⟨*symbol*⟩`}`

Using this macro, you can add a new checklist type. The name of the type, i.e., the name that can be used as argument to the `CheckList` environment, is specified by ⟨*type*⟩. The basic symbol of entries belonging to this checklist type will be ⟨*symbol*⟩ (e.g., an empty box or circle). All status-symbols (see Section 4.1) are drawn on top of ⟨*symbol*⟩. Note that the typed-checklist package uses this macro also for creating each of the four default checklist types.

`\CheckListAddStatus{`⟨*types*⟩`}{`⟨*status*⟩`}{`⟨*isclosed*⟩`}{`⟨*symbol*⟩`}`

Key	Description and Possible Values	Default
`layout`	This selects the default checklist layout. Allowed values are all known layout names, including the predefined '`list`', '`table`', '`hidden`'. In `list` layout, each entry is a list item. In `table` layout, each entry is a row and the checklist is a table (see Section 6.2 for how to change which table environment is used). The `hidden` layout does not display the checklist and its entries.	`list`
`input-dates`	This option specifies the format of deadlines that checklist entries expect. Allowed values are '`d.m.y`', '`m/d/y`', and '`y-m-d`' – with the intuitive meaning.	`d.m.y`
`output-dates`	This option specifies the format in which deadline dates are displayed. Allowed values are: '`d.m.y`', '`m/d/y`', '`y-m-d`': These format dates in the indicated order of day, month, and year. '`d.m.yy`', '`m/d/yy`', '`yy-m-d`': These are analogous to their counterparts with a single 'y', but use a two-digit display of the year (i.e., the century is stripped away). '`d.m.`', '`m/d`', '`m-d`': These format dates in the indicated order, showing only month and day of the month. '`same`': With `same`, deadlines are output in the same format in which they are specified. '`datetime`': With `datetime`, the datetime2 package is used for displaying deadlines. The package must be loaded manually.	`same`
`strict-dates`	This option specifies whether deadlines must adhere to the input date format (as specified via the `input-dates` key) or can deviate. Allowed values are '`true`' and '`false`'.	`false`

Table 1: Options for `CheckList` environments (and `\CheckListSet`)

☑ Y1K problems are resolved.

◯ Y2K problems are resolved. . . . . *(John)*

Status	Description	Who	Deadline
▶	(Task I) Repair all programs	John	
☐	Just turn off all computers, if Task I fails	Mankind	December 31, 1999

? Y65K problems are resolved.

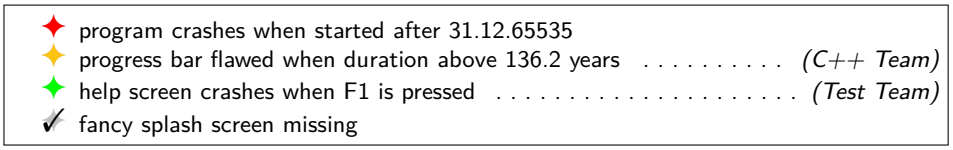
- ☐ (Task II) Build Y65K-proof time machine.
- ☐ Use time machine from Task II if problem persists.

```
\DTMsetregional% remember \usepackage{datetime2}
\CheckListSet{strict-dates,input-dates=d.m.y,output-dates=same}
\begin{CheckList}{Goal}
  \Goal[deadline=31.12.999]{achieved}{Y1K problems are resolved.}
  \Goal[who=John,deadline=31.12.1999]{open}{Y2K problems are resolved.}
    \begin{CheckList}[layout=table,output-dates=datetime]{Task}
      \Task[who=John,label=Fix1]{started}{Repair all programs}
      \Task[who=Mankind,deadline=31.12.1999]
          {open}{Just turn off all computers, if \ref{Fix1} fails}
    \end{CheckList}
  \Goal[deadline=31.12.65535]{unclear}{Y65K problems are resolved.}
    \begin{CheckList}[strict-dates=false,output-dates=m/d/y]{Task}
      \Task[deadline=$\approx 2500AD$,label=TM]
          {open}{Build Y65K-proof time machine.}
      \Task[deadline=31.12.65535]
          {open}{Use time machine from \ref{TM} if problem persists.}
    \end{CheckList}
\end{CheckList}
```

Listing 1: Comprehensive checklist example

Using this macro, you can add a new checklist entry status for selected checklist types. The name of the status to define is specified by the ⟨*status*⟩ argument. The checklist types to which the status is added, are provided by the ⟨*types*⟩ argument, a comma-separated list. The ⟨*symbol*⟩ is LaTeX code of a symbol that is put on top of the checklist type's symbol. The ⟨*isclosed*⟩ parameter must be one of `true` or `false`. A value of `true` indicates that the status of the entry corresponds to the entry being closed. This particularly means that no warning will be shown if the deadline of an entry with this status is passed. A value of `false` for ⟨*isclosed*⟩ indicates that the ⟨*status*⟩ corresponds to the entry not yet being closed. Note that the typed-checklist package uses this macro also for creating the provided states of the four default checklist types.

Example The following example shows how to define a 'bug' type.




```
\CheckListAddType{Bug}{\textcolor{lightgray}{\FourStar}}
\CheckListAddStatus{Bug}{new}{false}{\textcolor{red}{\FourStar}}
\CheckListAddStatus{Bug}{assigned}{false}{\textcolor{yellow!75!red}{\FourStar}}
\CheckListAddStatus{Bug}{resolved}{true}{\textcolor{green}{\FourStar}}
\CheckListAddStatus{Bug}{closed}{true}{\Checkmark}

\begin{CheckList}{Bug}
  \Bug{new}{program crashes when started after 31.12.65535}
  \Bug[who=C++ Team]{assigned}{progress bar flawed when duration above 136.2 years}
  \Bug[who=Test Team]{resolved}{help screen crashes when F1 is pressed}
  \Bug{closed}{fancy splash screen missing}
\end{CheckList}
```

4.2 Defining Checklist Layouts

`\CheckListDeclareLayout{`⟨*name*⟩`}{`⟨*fields*⟩`}{`⟨*begin*⟩`}{`⟨*end*⟩`}`

Using this macro, you can add a new checklist layout. The ⟨*begin*⟩ and ⟨*end*⟩ part is similar to a `\newenvironment`. The ⟨*fields*⟩ must be a comma-separated list of field names. A field name can be one of the following:

1. the name of an entry property (e.g., 'status', 'description', 'deadline', or 'who'),
2. the concatenation of multiple entry properties, separated by a '+' (e.g., 'deadline+status'), or
3. a fresh name that does not correspond to an entry property.

When one or multiple entry properties are referenced in a field name (cases 1 and 2), then the ⟨*code*⟩ argument to `\CheckListDefineFieldFormat` gets the properties' values as arguments when invoked.

`\CheckListDefineFieldFormat{`⟨*layout*⟩`}{`⟨*field*⟩`}{`⟨*code*⟩`}`

After the new type has been added, for each field in the comma-separated ⟨*fields*⟩, this macro must be used to define how a field is formatted. The ⟨*code*⟩ can take one or more arguments. If the ⟨*field*⟩ does not contain a '+', the ⟨*code*⟩ can take one argument, through which the value of the respective entry property is passed to ⟨*code*⟩. If ⟨*field*⟩ concatenates multiple property names with a '+', then the number of arguments equals the number of names in ⟨*field*⟩ and the properties are passed in the given order.

`\CheckListExtendLayout{`⟨*name*⟩`}{`⟨*base*⟩`}{`⟨*fields*⟩`}`

Using this macro, you can extend an existing checklist layout. Afterwards, the layout ⟨*name*⟩ is available. This layout takes the ⟨*begin*⟩ and ⟨*end*⟩ code from the ⟨*base*⟩ layout. Moreover, all fields defined by the ⟨*base*⟩ layout can be used in the ⟨*fields*⟩ parameter of the new layout. However, additional fields can be defined and the format of the fields for the new layout can be overwritten via `\CheckListDefineFieldFormat`.

Auxiliary Macros The following macros can be used in the definition of field formats.

`\CheckListStatusSymbol{`⟨*status*⟩`}`

The macro expands to the defined symbol for the given ⟨*status*⟩, i.e., the overlay between the checklist type's base symbol and the entry status' symbol.

`\CheckListSigned[`⟨*core*⟩`]{`⟨*text*⟩`}`

The macro displays ⟨*text*⟩ in a right-aligned fashion with a dotted leader to ⟨*text*⟩. This is similar to the display of page numbers in some table of content formats. The display takes place only if ⟨*text*⟩ is non-empty. If ⟨*core*⟩ is given, ⟨*core*⟩ is instead used in the emptiness check.

`\CheckListDefaultLabel{`⟨*label*⟩`}`

This macro sets ⟨*label*⟩ as the label for the current entry, based on the default checklist counter. It corresponds to a `\refstepcounter` on the checklist counter and a subsequent `\label{`⟨*label*⟩`}`.

`\CheckListDisplayDeadline{`⟨*status*⟩`}{`⟨*deadline*⟩`}`

This macro displays ⟨*deadline*⟩ depending on the given entry's ⟨*status*⟩ and the current date. Internally, for highlighting the deadline, the following macro is used, which can be redefined with `\renewcommand` to change the deadline highlighting.

`\CheckListHighlightDeadline{`⟨*closed?*⟩`}{`⟨*passed?*⟩`}{`⟨*deadline*⟩`}`

This macro formats ⟨*deadline*⟩ depending on whether the corresponding checklist entry is ⟨*closed?*⟩ (`true` or `false`) and whether ⟨*deadline*⟩ has already ⟨*passed?*⟩ (`true` or `false`).

Example The following example shows how to define an alternative list format.

1. Y1K problems are resolved. 31.12.999
2. John: Y2K problems are resolved. 31.12.1999
   (a) John: Repair all programs
   (b) Mankind: Just turn off all computers, if Task 2a fails 31.12.1999

```
\CheckListDeclareLayout{enumlist}{label,who,status,description+deadline+status}
  {\bgroup\topsep=\medskipamount\itemsep=0pt\enumerate}
  {\endenumerate\egroup}
\CheckListDefineFieldFormat{enumlist}{label}{\item\label{#1}}
\CheckListDefineFieldFormat{enumlist}{who}{\ifstrempty{#1}{}{#1: }}
\CheckListDefineFieldFormat{enumlist}{status}{\normalmarginpar\marginnote
  {\CheckListStatusSymbol{#1}}}
\CheckListDefineFieldFormat{enumlist}{description+deadline+status}
  {#1\CheckListSigned[#2]{\CheckListDisplayDeadline{#3}{#2}}}

\begin{CheckList}[layout=enumlist]{Goal}
  \Goal[deadline=31.12.999]{achieved}{Y1K problems are resolved.}
  \Goal[who=John,deadline=31.12.1999]{open}{Y2K problems are resolved.}
    \begin{CheckList}{Task}
      \Task[who=John,label=Fix1]{started}{Repair all programs}
      \Task[who=Mankind,deadline=31.12.1999]
          {open}{Just turn off all computers, if Task~\ref{Fix1} fails}
    \end{CheckList}
\end{CheckList}
```

4.3 Adding Entry Options

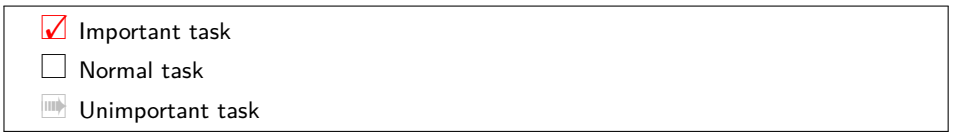
Checklist entries can be augmented by more than the default fields. Values for these additional fields can be specified as entry options.

`\CheckListAddEntryOption{`⟨*name*⟩`}{`⟨*default*⟩`}`

This macro introduces a new entry option named ⟨*name*⟩ and with the given ⟨*default*⟩ value. The newly introduced option can then be provided to a checklist entry in the same way as the pre-defined options “who” and “label”.

When an entry option is defined, by default it is not displayed. Hence, when introducing a new entry option, one should consider defining a new checklist layout that makes use of the entry option.

The following example shows how to extend a layout for incorporating a custom-defined priority field.




```
\CheckListAddEntryOption{priority}{M}
\usepackage{xcolor}
\colorlet{priocolor-H}{red}
\colorlet{priocolor-M}{black}
\colorlet{priocolor-L}{lightgray}
\CheckListExtendLayout{priolist}{list}{priority+status,label,description,
                                       who,deadline+status,END}
\CheckListDefineFieldFormat{priolist}{priority+status}{%
  \item[{\normalfont\textcolor{priocolor-#1}{\CheckListStatusSymbol{#2}}}]}

\begin{CheckList}[layout=priolist]{Task}
  \Task[priority=H]{done}{Important task}
  \Task{open}{Normal task}
  \Task[priority=L]{started}{Unimportant task}
\end{CheckList}
```

5 Filtering Checklists

Filtering out certain checklist entries based on their properties can help keeping the focus on the relevant entries. For this purpose, typed-checklist allows one to specify filtering code.

5.1 Setting Basic Filters

`\CheckListFilterClosed[`⟨*types*⟩`]`

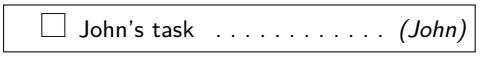
This macro sets up a filter that *hides* all checklist entries whose status is closed. Through the optional ⟨*types*⟩ argument, a comma-separated list of checklist types can be specified to which the filter shall be applied. By default, the filter is applied to all defined checklist types.

```
\CheckListFilterClosed
\begin{CheckList}{Task}
  \Task{open}{Open task}
  \Task{started}{Started task}
  \Task{done}{Done task}
  \Task{dropped}{Dropped task}
\end{CheckList}
```

`\CheckListFilterValue[`⟨*types*⟩`]{`⟨*field*⟩`}{`⟨*value*⟩`}`

This macro sets up a filter that *hides* all checklist entries whose ⟨*field*⟩ has a value that is unequal ⟨*value*⟩.

```
\CheckListFilterValue{who}{John}
\begin{CheckList}{Task}
  \Task[who=John]{open}{John's task}
  \Task[who=Mary]{open}{Mary's task}
\end{CheckList}
```



`\CheckListFilterDeadline[`⟨*types*⟩`]{`⟨*comp*⟩`}{`⟨*date*⟩`}{`⟨*filter-inv*⟩`}`

This macro sets up a filter that filters out checklist entries by their deadline. Only those entries are preserved whose deadline is before (if ⟨*comp*⟩ equals '<'), equal (if ⟨*comp*⟩ equals '='), or after (if ⟨*comp*⟩ equals '>') the given ⟨*date*⟩. The ⟨*date*⟩ *must* be in the format selected for input dates (see the `input-dates` option). If ⟨*filter-inv*⟩ is `true`, then checklist entries whose deadline does not obey the format for input dates are filtered out. Otherwise, if ⟨*filter-inv*⟩ is `false`, these checklist entries are not filtered out.

☐ John's task . . . . . *(John)*

☐ Other task (second time)

```
\CheckListFilterDeadline{<}{01.01.2019}{true}
\begin{CheckList}{Task}
  \Task[who=John,deadline=09.11.1989]{open}{John's task}
  \Task[who=Mary,deadline=01.01.2019]{open}{Mary's task}
  \Task[deadline=TBD]{open}{Other task (first time)}

  \CheckListFilterDeadline{<}{01.01.2019}{false}
  \Task[deadline=TBD]{open}{Other task (second time)}
\end{CheckList}
```

5.2 Combining and Resetting Filters

When multiple filter macros are used, the filters are applied one after another to each checklist entry until a filter filters out the entry. Consequentially, all applied filters are combined conjunctively, i.e., only those checklist entries are displayed that satisfy all filters.

When two filters are set up that affect the exact same fields of checklist entries (of the same type), then only the last of these filters becomes effective. The following example demonstrates this as well as the conjunction of filters.

```
\CheckListFilterValue{who}{John}
\CheckListFilterValue[Task]{status}{done}
\CheckListFilterValue[Goal]{status}{achieved}
\CheckListFilterValue{who}{Mary}
\begin{CheckList}{Goal}
  \Goal[who=Mary]{achieved}{Mary's goal}
    \begin{CheckList}{Task}
      \Task[who=John]{done}{John's task}
      \Task[who=Mary]{done}{Mary's task}
      \Task[who=Mary]{open}{Mary's open task}
    \end{CheckList}
\end{CheckList}
```

☑ Mary's goal . . . . . *(Mary)*

☑ Mary's task . . *(Mary)*

Filters are local to the LaTeX group in which they are set up. In particular, if a filter is set up inside an environment, then the filter is no longer effective after the environment.

```
\begin{CheckList}{Goal}
  \Goal[who=Mary]{achieved}{Mary's goal}
    \begin{CheckList}{Task}
      \CheckListFilterValue{who}{Mary}
      \Task[who=Mary]{done}{Mary's task}
      \Task[who=John]{done}{John's task}
    \end{CheckList}
  \Goal[who=John]{achieved}{John's goal}
\end{CheckList}
```

☑ Mary's goal . . . . . *(Mary)*
☑ Mary's task . . *(Mary)*
☑ John's goal . . . . . *(John)*

`\CheckListFilterReset[`⟨*types*⟩`]`

This macro removes all filters. If ⟨*types*⟩ are given, then only the filters for the checklist types in the comma-separated ⟨*types*⟩ are removed.

```
\CheckListFilterValue{who}{John}
\begin{CheckList}{Task}
  \Task[who=John]{open}{John's task (1)}
  \Task[who=Mary]{open}{Mary's task (1)}
\end{CheckList}
\CheckListFilterReset[Task]
\begin{CheckList}{Task}
  \Task[who=John]{open}{John's task (2)}
  \Task[who=Mary]{open}{Mary's task (2)}
\end{CheckList}
\begin{CheckList}{Goal}
  \Goal[who=Mary]{achieved}{Mary's goal (3)}
  \Goal[who=John]{achieved}{John's goal (3)}
\end{CheckList}
```

☐ John's task (1) . . . . . *(John)*
☐ John's task (2) . . . . . *(John)*
☐ Mary's task (2) . . . *(Mary)*
☑ John's goal (3) . . . . . *(John)*

5.3 The Generic Filter Interface

Filters can also be set up programmatically.

`\CheckListSetFilter[`⟨*types*⟩`]{`⟨*fields*⟩`}{`⟨*filter-code*⟩`}`

This macro sets up the ⟨*filter-code*⟩ for a set of ⟨*fields*⟩. The ⟨*fields*⟩ must be given as a '+'-separated list of field names, e.g., "`status+who`". The ⟨*filter-code*⟩ may contain as many positional parameters (`#1`, . . .) as there are fields names in ⟨*fields*⟩. When a checklist entry is about to be displayed, the ⟨*filter-code*⟩ is evaluated, obtaining as arguments the entry's field values. By default (without any filter set up), all entries are displayed. To disable the display of an entry, the ⟨*filter-code*⟩ can use `\togglefalse{display}`. If ⟨*types*⟩ are given (as a comma-separated list), then the ⟨*filter-code*⟩ is applied only to checklists of a type in the list.

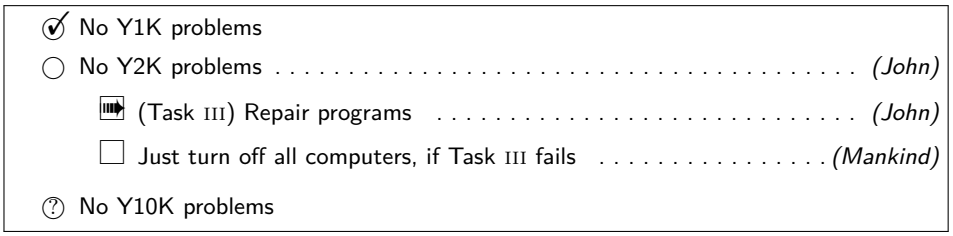
Examples for how to use the macro can be found in the implementation, e.g., of the macros `\CheckListFilterClosed` and `\CheckListFilterValue`.

6 Checklists and Other Packages

6.1 asciilist

The typed-checklist package can be combined with the asciilist package in the sense that a checklist can be defined within an `AsciiList` environment. The typed-checklist package provides a syntax for this when the package is loaded with the

withAsciilist=true option. The syntax is illustrated with the following snippet, a transformed version of the example in Section 3.3:




```
\usepackage[withAsciilist=true]{typed-checklist}
\begin{AsciiList}[GoalList,TaskList]{-,*}
  - achieved[deadline=31.12.999]: No Y1K problems
  - open[who=John,deadline=31.12.1999]: No Y2K problems
    * started[who=John,label=Fix2]: Repair programs
    * open[who=Mankind,deadline=31.12.1999]:%
       Just turn off all computers, if \ref{Fix2} fails
  - unclear[deadline=31.12.9999]: No Y10K problems
\end{AsciiList}
```

For each checklist type ⟨*type*⟩ (added by \CheckListAddType), an AsciiList environment ⟨*type*⟩List is automatically created.

Note that currently, a checklist entry in an AsciiList environment must fit into a single line *or* each except for the last line is ended with a percent char (as in the above example). Note also that the table layout does not work within an AsciiList environment.

6.2 Table Packages

The table layout by default uses the xltabular package for layouting the tables. The default can be changed through the tablepkg package option. The following values are available:

ltablex	This option uses the ltablex package.
tabularx	This option uses the tabularx package from the LaTeX core. When using this table type, keep in mind that tabularx tables must fit onto a single page.
xltabular	This option uses the xltabular package, a successor of ltablex.

7 Related Packages

The following LaTeX packages provide related functionalities to the typed-checklist package.

todo:
The package allows for typesetting "to-dos", i.e., tasks in some sense, in a simple way with customizable display. The three main conceptual differences between todo and typed-checklist are:

1. todo does not distinguish between different types (such as goals and tasks);
2. todo does not allow one to provide a status for a to-do and rather assumes that done to-dos are simply removed from the document;
3. todo aims at specifying tasks for the document into which the to-dos are placed, while typed-checklist aims at typesetting checklists whose entries are for more general kinds of projects.

easy-todo:
The package is similar in spirit to the todo package and shares the main differences to the typed-checklist package.

todonotes:
The package is similar in spirit to todo and easy-todo, but provides more formatting options for the to-dos.

pgfgantt:
The package allows one to create Gantt charts, i.e., graphical displays of activities and milestones with a focus on time frames. The package allows one to structure the activities into groups. In that sense, there are certain similarities between the packages. The main conceptual difference to typed-checklist is the form of presentation (time-centric Gantt chart vs. text-centric lists).

8 Limitations and Future Work

- In `twoside` documents, deadlines are currently displayed in the left margin on even pages. The default layout (`list`) does not look good then. This should be repaired. The same problem is with checklist entry labels, which are displayed on the other side.
- In deadlines, the full year (including century) must be provided for the colored highlighting to work. Future versions could check for a two-digit year and automatically prepend "20" for the century.
- The package automatically adds the pre-defined checklist types and states, which might have two draws for some users: firstly, this adds a dependency on symbol packages, which might not work well together with some fonts; secondly, some users might prefer other definitions of the standard checklist types. To improve the situation, the package could offer an option for disabling the definition of the standard checklist types. Concerning the symbols packages, typed-checklist could also reduce the set of used packages or even draw all symbols itself.
- The package displays checklist entries in the ordering in which they are listed in the LaTeX sources. Automatic sorting of checklist entries, for instance by deadline or future fields like priority/importance, might make the package even more useful for bigger checklists. The implementation of the feature could be done for example as discussed on stackexchange.

9 Pre-defined Checklist Types and States

```
\paragraph{Goals}
\begin{CheckList}{Goal}
  \Goal{open}{open goal}
  \Goal{dropped}{dropped goal}
  \Goal{unclear}{unclear goal}
  \Goal{achieved}{achieved goal}
\end{CheckList}

\paragraph{Tasks}
\begin{CheckList}{Task}
  \Task{open}{open task}
  \Task{dropped}{dropped task}
  \Task{unclear}{unclear task}
  \Task{started}{started task}
  \Task{done}{done task}
\end{CheckList}

\paragraph{Artifacts}
\begin{CheckList}{Artifact}
  \Artifact{missing}{missing artifact}
  \Artifact{dropped}{dropped artifact}
  \Artifact{unclear}{unclear artifact}
  \Artifact{incomplete}{incomplete artifact}
  \Artifact{available}{available artifact}
\end{CheckList}

\paragraph{Milestones}
\begin{CheckList}{Milestone}
  \Milestone{open}{open milestone}
  \Milestone{achieved}{achieved milestone}
\end{CheckList}
```

Goals

- ◯ open goal
- ⊗ dropped goal
- ? unclear goal
- ☑ achieved goal

Tasks

- ☐ open task
- ☒ dropped task
- ? unclear task
- started task
- ☑ done task

Artifacts

- △ missing artifact
- dropped artifact
- unclear artifact
- incomplete artifact
- available artifact

Milestones

- ☆ open milestone
- ★ achieved milestone

10 Implementation

10.1 Basic Package Dependencies

We use the xkeyval package for declaring package options as well as for option lists of entry types.

```
\RequirePackage{xkeyval}
```

We use the etoolbox package for simpler handling of lists.

```
\RequirePackage{etoolbox}
```

We use colors for deadlines, for instance.

```
\RequirePackage{xcolor}
```

10.2 Options

10.2.1 Checklist Options

In the following, we define the possible options for a checklist.

```
\define@key[tchklst]{GlobalListOptions}{layout}{%
  \ifinlist{#1}{\tchklst@ChecklistLayouts}{}{%
    \PackageError{typed-checklist}{%
      `#1' not a known checklist layout}
      {Known layouts are:\forlistloop{ }{\tchklst@@CheckListLayouts}}}%
  \def\tchklst@@layout{#1}}
\define@key[tchklst]{GlobalListOptions}{input-dates}{%
  \ifinlist{#1}{\tchklst@@InputDateFormats}{}{%
    \PackageError{typed-checklist}{%
      `#1' not a known input date format}
      {Known formats are:\forlistloop{ }{\tchklst@@InputDateFormats}}}%
  \letcs\tchklst@inputdate@order{tchklst@dateorder@#1}%
  \letcs\tchklst@inputdate@sep{tchklst@dateformat@sep@#1}}
\define@key[tchklst]{GlobalListOptions}{output-dates}{%
  \ifinlist{#1}{\tchklst@@OutputDateFormats}{}{%
    \PackageError{typed-checklist}{%
      `#1' not a known output date format}
      {Known formats are:\forlistloop{ }{\tchklst@@OutputDateFormats}}}%
  \letcs\tchklst@@dateoutput@use{tchklst@dateoutput@use@#1}}
\define@boolkey[tchklst]{GlobalListOptions}{strict-dates}[true]{%
  \ifbool{tchklst@GlobalListOptions@strict-dates}
    {\let\tchklst@@faileddate=\tchklst@DateFailStrict}
    {\let\tchklst@@faileddate=\tchklst@DateFailLax}}
```

10.2.2 Checklist Entry Options

\CheckListAddEntryOption

The \CheckListAddEntryOption{⟨*option*⟩}{⟨*default*⟩} macro declares a new ⟨*option*⟩ that can be used when defining checklist entries. An option always comes with a ⟨*default*⟩ value.

```
\newcommand*\CheckListAddEntryOption[2]{%
  \define@cmdkey[tchklst]{Entry}{#1}[#2]{}%
  \presetkeys[tchklst]{Entry}{#1}{}}
```

In the following, we define a basic default set of possible options for a checklist entry.

```
\CheckListAddEntryOption{who}{}
\CheckListAddEntryOption{deadline}{}
\CheckListAddEntryOption{label}{}
```

10.3 Setting Options Globally

\CheckListSet The \CheckListSet{⟨*options-list*⟩} sets global options for the typed-checklist package.

```
\newcommand\CheckListSet[1]{%
  \setkeys[tchklst]{GlobalListOptions}{#1}}
```

\CheckListDefaultLayout The \CheckListDefaultLayout{⟨*layout*⟩} macro sets the default layout for all CheckList environments that do not set the layout option explicitly. This macro is obsoleted by the \CheckListSet macro introduced in v2.0 of the package.

```
\newcommand*\CheckListDefaultLayout[1]{%
  \CheckListSet{layout={#1}}}
```

10.4 Checklist Types

In the following, we implement the existing types of checklists as well as the macros for declaring new types.

\tchklst@ChecklistTypes The \tchklst@ChecklistTypes collects the list of known checklist types. Initially, the list is empty.

```
\newcommand*\tchklst@ChecklistTypes{}
```

\CheckListAddType The \CheckListAddType{⟨*type*⟩}{⟨*symbol*⟩} adds a new checklist type with name ⟨*type*⟩ to the list of known checklist types. The basic symbol of entries belonging to this checklist type will be ⟨*symbol*⟩ (e.g., an empty box or circle).

```
\newcommand*\CheckListAddType[2]{%
```

Add new type to existing list, if the type is not already known.

```
  \ifinlist{#1}{\tchklst@ChecklistTypes}{%
    \PackageError{typed-checklist}{%
      Checklist type `#1' already defined}{}}{}
  \listadd\tchklst@ChecklistTypes{#1}%
```

Save the symbol of the new type.

```
  \csdef{tchklst@ChecklistTypeSym@#1}{#2}%
```

Create an initially empty list of possible states that entries of the type can have, and an empty list of filters for the type.

```
  \csdef{tchklst@ChecklistStates@#1}{}%
  \csdef{tchklst@ChecklistFilters@#1}{}%
```

Finally, invoke all hooks for new types of checklists.

```
  \def\do##1{##1{#1}}%
  \dolistloop\tchklst@@addtype@hooks}
```

\tchklst@@addtype@hooks This is an etoolbox list of single-argument macros for hooking into the registration of new checklist types.

```
\newcommand*\tchklst@@addtype@hooks{}
```

\tchklst@IntroduceTypeHook The \tchklst@IntroduceTypeHook{⟨*cmd*⟩} macro introduces ⟨*cmd*⟩ for all existing checklist types (first code line) as well as for all checklist types defined afterwards (second code line).

```
\newcommand*\tchklst@IntroduceTypeHook[1]{%
  \forlistloop{#1}{\tchklst@ChecklistTypes}%
  \listgadd\tchklst@@addtype@hook{#1}}
```

\tchklst@aux@OargAfter The \tchklst@aux@OargAfter{⟨*macro-use*⟩}[⟨*opt-arg*⟩] macro inserts an optional argument, ⟨*opt-arg*⟩, into a ⟨*macro-use*⟩, where the ⟨*macro-use*⟩ may have multiple mandatory arguments but no optional argument. The ⟨*opt-arg*⟩ is optional, i.e., if it is not provided, then ⟨*macro-use*⟩ is taken as is.

Example use: \tchklst@aux@OargAfter{\cite{foo}}[page 9] would expand first to \tchklst@aux@OargAfter@ii{page 9}\cite{foo} and, finally, to \cite[page 9]{foo}.

```
\newcommand\tchklst@aux@OargAfter[1]{%
  \@ifnextchar[{\tchklst@aux@OargAfter@i{#1}}{#1}}
\long\def\tchklst@aux@OargAfter@i#1[#2]{%
  \tchklst@aux@OargAfter@ii{#2}#1}
\newcommand\tchklst@aux@OargAfter@ii[2]{%
  #2[#1]}
```

\tchklst@CheckType The \tchklst@CheckType{⟨*type*⟩} is a convenience macro for checking whether the checklist type ⟨*type*⟩ is defined. This macro yields an error with a simple message if ⟨*type*⟩ is not defined.

```
\newcommand*\tchklst@CheckType[1]{%
  \ifinlist{#1}{\tchklst@ChecklistTypes}{}{%
    \PackageError{typed-checklist}%
      {Unknown checklist type `#1'}
      {Known types are:\forlistloop{ }{\tchklst@ChecklistTypes}}}}
```

10.5 Checklist Entry States

In the following, we implement the existing status possibilities of the individual checklist types as well as macros for declaring a new status.

\CheckListAddStatus The \CheckListAddStatus{⟨*types*⟩}{⟨*status*⟩}{⟨*isclosed*⟩}{⟨*symbol*⟩} macro declares a new ⟨*status*⟩ for a given comma-separated list of checklist ⟨*types*⟩. The ⟨*symbol*⟩ is LaTeX code of a symbol that is put on top of the checklist type's symbol. The ⟨*isclosed*⟩ parameter must be one of `true` or `false`. A value of `true` indicates that the status of the entry corresponds to the entry being closed. This particularly means that no warning will be shown if the deadline of an entry with this status is passed. A value of `false` for ⟨*isclosed*⟩ indicates that the ⟨*status*⟩ corresponds to the entry not yet being closed.

```
\newcommand*\CheckListAddStatus[4]{%
```

We loop over all the checklist ⟨*types*⟩ given.

```
\forcsvlist
```

In the following line, the actual type parameter is added last by the `\forcsvlist` macro.

```
    {\tchklst@AddStatus{#2}{#3}{#4}}%
    {#1}}%
```

`\tchklst@AddStatus` The `\tchklst@AddStatus{`⟨*status*⟩`}{`⟨*isclosed*⟩`}{`⟨*symbol*⟩`}{`⟨*type*⟩`}` has the same parameters (in different ordering) and intention as the `\CheckListAddStatus` macro, except that it assumes a single ⟨*type*⟩ instead of a type list. This macro is used internally by `\CheckListAddStatus`.

```
\newcommand*\tchklst@AddStatus[4]{%
```

Some argument checking up front.

```
  \tchklst@CheckType{#4}%
  \ifinlistcs{#1}{tchklst@ChecklistStates@#4}{%
    \PackageError{typed-checklist}{%
      #4-checklist state ‘#1’ already defined}{}}{}
```

Register the status for the checklist type.

```
  \listcsadd{tchklst@ChecklistStates@#4}{#1}%
```

Register the status symbol and “isclosed”.

```
  \expandafter\def\csname tchklst@isclosed@#4@#1\endcsname{#2}%
  \expandafter\def\csname tchklst@sym@#4@#1\endcsname{#3}}
```

`\tchklst@CheckTypeStatus` The `\tchklst@CheckTypeStatus{`⟨*type*⟩`}{`⟨*status*⟩`}` is a convenience macro for checking whether the checklist entry status ⟨*status*⟩ is defined for checklist type ⟨*type*⟩. This macro yields an error with a simple message if ⟨*status*⟩ is not defined.

```
\newcommand*\tchklst@CheckTypeStatus[2]{%
  \ifinlistcs{#2}{tchklst@ChecklistStates@#1}{}{%
    \PackageError{typed-checklist}%
      {Unknown #1-checklist entry status ‘#2’}%
      {Known states are:\forlistcsloop{ }{tchklst@ChecklistStates@#1}}}}
```

`\CheckListStatusSymbol` The `\CheckListStatusSymbol{`⟨*status*⟩`}` is a convenience macro for obtaining the symbol for a particular ⟨*status*⟩ of the current checklist's type.

```
\newcommand*\CheckListStatusSymbol[1]{%
  \tchklst@symbolcombine{\csuse{tchklst@sym@\tchklst@cur@type @#1}}%
    {\csuse{tchklst@ChecklistTypeSym@\tchklst@cur@type}}}
```

`\tchklst@symbolcombine` The `\tchklst@symbolcombine{`⟨*symbol1*⟩`}{`⟨*symbol2*⟩`}` macro combines two symbols, ⟨*symbol1*⟩ and ⟨*symbol2*⟩.

```
\newcommand*\tchklst@symbolcombine[2]{{%
  \setbox0\hbox{#2}%
  \copy0\llap{\hbox to \wd0{\hss\smash{#1}\hss}}}}
```

`\CheckListIfClosed` The `\CheckListIfClosed{`⟨*status*⟩`}{`⟨*iftrue*⟩`}{`⟨*iffalse*⟩`}` macro expands to ⟨*iftrue*⟩, if the ⟨*status*⟩ of an entry in the current checklist is a “closed” one (see the documentation for `\CheckListAddStatus` for details). Otherwise, the macro expands to ⟨*iffalse*⟩.

```
\newcommand*\CheckListIfClosed[1]{%
  \csname if\csname tchklst@isclosed@\tchklst@cur@type @#1\endcsname
            \endcsname
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi}
```

10.6 Checklist Layouts

\tchklst@ChecklistLayouts

The \tchklst@ChecklistLayouts collects the list of known checklist layouts. Initially, the list is empty.

```
\newcommand*\tchklst@ChecklistLayouts{}
```

\CheckListDeclareLayout

The \CheckListDeclareLayout{⟨*name*⟩}{⟨*fields*⟩}{⟨*begin*⟩}{⟨*end*⟩} macro declares a new checklist layout with the given ⟨*name*⟩. At the begin and end of the checklist, the ⟨*begin*⟩ and, respectively, ⟨*end*⟩ code is executed. The ⟨*fields*⟩ parameter must be a comma-separated list of field names. The fields will be displayed for each checklist entry in the order given by ⟨*fields*⟩, where the format for the display must be declared using \CheckListDefineFieldFormat.

```
\newcommand*\CheckListDeclareLayout[4]{%
```

Add new layout to existing list, if the layout is not already known.

```
  \ifinlist{#1}{\tchklst@ChecklistLayouts}{%
    \PackageError{typed-checklist}{%
      Checklist layout `#1' already declared}{}}{}
  \listadd\tchklst@ChecklistLayouts{#1}%
```

Save the ⟨*fields*⟩ list of the new layout.

```
  \csdef{tchklst@ChecklistLayoutFields@#1}{}%
  \forcsvlist{\listcsadd{tchklst@ChecklistLayoutFields@#1}}{#2}%
```

Save the ⟨*begin*⟩ and ⟨*end*⟩ code of the new layout.

```
  \csdef{tchklst@ChecklistLayoutBegin@#1}{#3}%
  \csdef{tchklst@ChecklistLayoutEnd@#1}{#4}}
```

\CheckListExtendLayout

The \CheckListExtendLayout{⟨*name*⟩}{⟨*base*⟩}{⟨*fields*⟩} macro declares a new checklist layout, ⟨*name*⟩, which inherits existing ⟨*fields*⟩ as well as the ⟨*begin*⟩ and ⟨*end*⟩ code from a given ⟨*base*⟩ layout.

```
\newcommand*\CheckListExtendLayout[3]{%
  \CheckListDeclareLayout{#1}{#3}%
    {\csuse{tchklst@ChecklistLayoutBegin@#2}}%
    {\csuse{tchklst@ChecklistLayoutEnd@#2}}%
```

Inherit all fields defined by the ⟨*base*⟩ layout.

```
  \def\do##1{%
    \ifcsdef{tchklst@ChecklistFormat@#2@##1}{%
      \csletcs{tchklst@ChecklistFormat@#1@##1}%
              {tchklst@ChecklistFormat@#2@##1}}{}}%
  \dolistcsloop{tchklst@ChecklistLayoutFields@#2}%
}
```

\CheckListDefineFieldFormat The \CheckListDefineFieldFormat{⟨*layout*⟩}{⟨*field*⟩}{⟨*code*⟩} macro defines the ⟨*code*⟩ to be used for displaying the given ⟨*field*⟩ (or fields) in a checklist of the given ⟨*layout*⟩. Multiple fields can be displayed by specifying ⟨*field*⟩ in the form $\langle field\rangle_1 + \ldots + \langle field\rangle_n$.

```
\newcommand\CheckListDefineFieldFormat[3]{%
  \tchklst@deffieldmacro{tchklst@ChecklistFormat@#1@#2}{#2}{#3}}
```

\tchklst@deffieldmacro The \tchklst@deffieldmacro{⟨*csname*⟩}{⟨*fields*⟩}{⟨*code*⟩} defines a command with name ⟨*csname*⟩ whose number of arguments equals the number of '+'-separated elements in ⟨*fields*⟩. The command then expands to ⟨*code*⟩, which can refer to the respective number of positional parameters.

```
\newcommand\tchklst@deffieldmacro[3]{%
  \begingroup
```

Get number of properties ('+'-separated) in ⟨*field*⟩ into \@tempcnta.

```
  \@tempcnta=0\relax
  \def\do##1{\advance\@tempcnta by 1\relax}%
  \tchklst@dopsvlist{#2}%
```

Next, use the above number for determining the number of arguments of the defined formatting macro, i.e., the number of positional parameters permitted in ⟨*code*⟩. The macro is first \undef'd such that \newcommand always succeeds.

```
  \edef\do{\endgroup
    \csundef{#1}%
    \noexpand\newcommand\expandonce{\csname #1\endcsname}%
      [\the\@tempcnta]{\unexpanded{#3}}}%
  \do}
```

\tchklst@usefieldmacro The \tchklst@usefieldmacro[⟨*use-cmd*⟩]{⟨*csname*⟩}{⟨*fields*⟩} macro takes the current values of the fields in the '+'-separated ⟨*fields*⟩ and applies them in the given order to ⟨*csname*⟩. This application is performed directly, if ⟨*use-cmd*⟩ is left at its default, or is otherwise provided as an argument to ⟨*use-cmd*⟩.

```
\newcommand\tchklst@usefieldmacro[3][\@firstofone]{%
  \begingroup
  \expandafter\def\expandafter\tchklst@@cmd\expandafter{%
    \csname #2\endcsname}%
  \def\do##1{\eappto\tchklst@@cmd{%
    {\csexpandonce{cmdtchklst@Entry@##1}}}}%
  \tchklst@dopsvlist{#3}%
  \expandafter\def\expandafter\tchklst@@cmd\expandafter{%
    \expandafter{\tchklst@@cmd}}%
  \preto\tchklst@@cmd{\endgroup#1}%
  \tchklst@@cmd}
```

\tchklst@CheckLayout The \tchklst@CheckLayout{⟨*layout*⟩} is a convenience macro for checking whether the checklist layout ⟨*layout*⟩ is defined. This macro yields an error with a simple message if ⟨*layout*⟩ is not defined. If a command is provided for the ⟨*layout*⟩, it is expanded.

```
\newcommand*\tchklst@CheckLayout[1]{%
  \xifinlist{#1}{\tchklst@ChecklistLayouts}{}{%
    \PackageError{typed-checklist}%
```

```
      {Unknown checklist layout `#1'}
      {Known layouts are:\forlistloop{ }{\tchklst@ChecklistLayouts}}}}
```

10.7 Entry Filters

\CheckListSetFilter
\tchklst@SetFilter

The `\CheckListSetFilter[`⟨*types*⟩`]{`⟨*fields*⟩`}{`⟨*code*⟩`}` macro defines a filter for the given ⟨*fields*⟩ of all types in the comma-separated list ⟨*types*⟩. The filtering code is ⟨*code*⟩, which may use positional parameters.

```
\newcommand*\CheckListSetFilter[3][*]{%
  \ifstrequal{#1}{*}
    {\forlistloop{\tchklst@SetFilter{#2}{#3}}{\tchklst@ChecklistTypes}}
    {\forcsvlist{\tchklst@SetFilter{#2}{#3}}{#1}}}
```

The `\tchklst@SetFilter{`⟨*fields*⟩`}{`⟨*code*⟩`}{`⟨*type*⟩`}` macro defines a filter for a single type.

```
\newcommand*\tchklst@SetFilter[3]{%
  \tchklst@CheckType{#3}%
  \ifinlistcs{#1}{tchklst@ChecklistFilters@#3}{}
    {\listcsadd{tchklst@ChecklistFilters@#3}{#1}}%
  \tchklst@deffieldmacro{tchklst@CheckListFilter@#3@#1}{#1}{#2}}
```

\CheckListFilterValue

The `\CheckListFilterValue[`⟨*types*⟩`]{`⟨*field*⟩`}{`⟨*value*⟩`}` macro filters out all checklist entries whose ⟨*field*⟩ is unequal ⟨*value*⟩, by using an `\ifstrequal` comparison.

```
\newcommand*\CheckListFilterValue[3][*]{%
  \CheckListSetFilter[#1]{#2}
    {\ifstrequal{##1}{#3}{}{\togglefalse{display}}}}
```

\CheckListFilterClosed

The `\CheckListFilterClosed[`⟨*types*⟩`]{`⟨*field*⟩`}{`⟨*value*⟩`}` macro filters out all checklist entries whose status is closed, by using `\CheckListIfClosed`.

```
\newcommand*\CheckListFilterClosed[1][*]{%
  \CheckListSetFilter[#1]{status}
    {\CheckListIfClosed{##1}{\togglefalse{display}}{}}}
```

\CheckListFilterDeadline

The `\CheckListFilterDeadline[`⟨*types*⟩`]{`⟨*comp*⟩`}{`⟨*refdate*⟩`}{`⟨*filter-inv*⟩`}` macro filters out all checklist entries whose deadline does not satisfy the comparison against ⟨*refdate*⟩ by operator ⟨*comp*⟩ (‘<’, ‘=’, ‘>’). The argument ⟨*filter-inv*⟩ must be either `true` or `false` and specifies whether deadlines that do not match the selected input deadline format are filtered out (`true`) or not (`false`).

```
\newcommand*\CheckListFilterDeadline[4][*]{%
```

First, pre-parse ⟨*refdate*⟩ such that it need not be parsed for each checklist entry.

```
  \bgroup
  \def\do##1##2##3##4{\egroup
```

Use the internal `\tchklst@DateCompare` macro to perform the date comparison based on the pre-parsed date of the `\do{`⟨*year*⟩`}{`⟨*month*⟩`}{`⟨*day*⟩`}` macro arguments.

```
    \CheckListSetFilter[#1]{deadline}
      {\tchklst@DateCompare{####1}{#2}{##1}{##2}{##3}
        {}{\togglefalse{display}}
```

```
        {\ifbool{#4}{\togglefalse{display}}{}\@gobble}}}
  \CheckListParseDate{#3}{\do}
```

If parsing ⟨*refdate*⟩ fails, we always fail like with strict input date parsing: Setting up a filter with an invalid date would not make sense.

```
    {\egroup\tchklst@DateFailStrict}}
```

\CheckListFilterReset The \CheckListFilterReset[⟨*types*⟩] macro resets the filters for all checklist types in the comma-separated list ⟨*types*⟩. If ⟨*types*⟩ is omitted or equals '*', then the filters for all checklist types are reset.

```
\newcommand*\CheckListFilterReset[1][*]{%
  \ifstrequal{#1}{*}
    {\forlistloop{\tchklst@ResetFilter}{\tchklst@ChecklistTypes}}
    {\forcsvlist{\tchklst@ResetFilter}{#1}}}
```

\tchklst@ResetFilter The \tchklst@ResetFilter{⟨*type*⟩} macro removes all filters (i.e., for all fields) from checklists of the given ⟨*type*⟩.

```
\newcommand*\tchklst@ResetFilter[1]{%
  \def\do##1{\csundef{tchklst@CheckListFilter@#1@##1}}%
  \dolistcsloop{tchklst@ChecklistFilters@#1}%
  \csdef{tchklst@ChecklistFilters@#1}{}}
```

10.8 Checklist and Entry Definition

CheckList The CheckList[⟨*options*⟩]{⟨*type*⟩} environment declares a new checklist.

```
\newenvironment{CheckList}[2][]{%
```

We check whether the provided ⟨*type*⟩ is known.

```
  \tchklst@CheckType{#2}%
```

Parse and check the options.

```
  \setkeys[tchklst]{GlobalListOptions}{#1}%
  \tchklst@CheckLayout{\tchklst@@layout}%
```

We store the type, layout, and fields of the checklist for use inside the list.

```
  \edef\tchklst@cur@type{#2}%
  \let\tchklst@cur@layout=\tchklst@@layout%
  \letcs\tchklst@cur@fields
    {tchklst@ChecklistLayoutFields@\tchklst@cur@layout}%
```

The following line declares the macro for the checklist entries, for example the \Goal macro for the ⟨*type*⟩ Goal.

```
  \cslet{#2}{\tchklst@entry}%
```

Start and end the actual checklist environment as defined by the layout.

```
  \csname tchklst@ChecklistLayoutBegin@\tchklst@cur@layout\endcsname
}{%
  \csname tchklst@ChecklistLayoutEnd@\tchklst@cur@layout\endcsname
}
```

\tchklst@entry The \tchklst@entry[⟨*options*⟩]{⟨*status*⟩}{⟨*description*⟩} macro defines a checklist entry with a given ⟨*status*⟩, a given ⟨*description*⟩, and possibly particular

⟨*options*⟩ (a comma-separated list of key-value pairs). See Section 10.2.2 for the list of available options.

```
\newcommand\tchklst@entry[3][]{%
  \begingroup
```

First check for a valid status. There is no need to check for a valid type, because the surrounding `CheckList` environment already does this.

```
  \tchklst@CheckTypeStatus{\tchklst@cur@type}{#2}%
```

Parse the options.

```
  \setkeys[tchklst]{Entry}{#1}%
```

Save status and description such that they can be accessed just like the options.

```
  \def\cmdtchklst@Entry@status{#2}%
  \def\cmdtchklst@Entry@description{#3}%
```

Now iterate through all filters for the current type until one filter turns the local `display` toggle to false.

```
  \newtoggle{display}\toggletrue{display}%
  \def\do##1{%
    \tchklst@usefieldmacro
      {tchklst@CheckListFilter@\tchklst@cur@type @##1}{##1}%
    \iftoggle{display}{}{\listbreak}}%
  \dolistcsloop{tchklst@ChecklistFilters@\tchklst@cur@type}%
```

Show the fields of the entry in the order they were given. The whole entry is first collected in a macro (`\tchklst@@entry`), such that individual field display code cannot leave the current LaTeX group (e.g., by advancing to the next table cell in table layout) and thereby void the entry option macros.

```
  \def\tchklst@@entry{\endgroup}%
  \iftoggle{display}{%
    \def\do##1{%
      \tchklst@usefieldmacro[\appto\tchklst@@entry]
        {tchklst@ChecklistFormat@\tchklst@cur@layout @##1}{##1}}%
    \dolistloop\tchklst@cur@fields}{}%
  \tchklst@@entry}
```

`\tchklst@dopsvlist` The `\tchklst@dopsvlist{`⟨*list*⟩`}` parses a '+'-separated list.

```
\DeclareListParser{\tchklst@dopsvlist}{+}
```

`\CheckListSigned` The `\CheckListSigned[`⟨*core*⟩`]{`⟨*text*⟩`}` macro is taken from Knuth's TeXbook with minor spacing modifications. See also http://tex.stackexchange.com/a/13761. The added optional ⟨*core*⟩ is the reference for checks whether ⟨*text*⟩ is empty: In case of emptiness, nothing is shown by the macro. If ⟨*core*⟩ is omitted, ⟨*text*⟩ itself is used in the emptiness check.

```
\newcommand\CheckListSigned{\@dblarg\tchklst@signed}
\newcommand\tchklst@signed[2][]{%
  \ifstrempty{#1}
    {\nobreak\hfill\null}
    {\leavevmode\unskip\nobreak\hfil\penalty50\hskip0.25em
     \hbox{}\nobreak\dotfill\hbox{#2}}}
```

10.9 Deadlines

The following code implements the parsing of deadlines and for comparing deadlines against the current date.

\CheckListDisplayDeadline The \CheckListDisplayDeadline{⟨*status*⟩}{⟨*deadline*⟩} formats a ⟨*deadline*⟩ dependent on the ⟨*status*⟩ and the current date.

```
\newcommand\CheckListDisplayDeadline[2]{%
```

Try to parse ⟨*deadline*⟩ as a date.

```
  \CheckListParseDate{#2}{\tchklst@DisplayDeadline@i}
%    \begin{macrocode}
    {\tchklst@firstoftwoargs\tchklst@@faileddate}
```

To each of the two former arguments (for successful and, respectively, failed parsing of the deadline), apply ⟨*deadline*⟩ as argument.

```
    {#1}}
```

\tchklst@firstoftwoargs The \tchklst@firstoftwoargs{⟨*cmd*⟩}{⟨*arg1*⟩}{⟨*arg2*⟩} macro applies ⟨*cmd*⟩ to ⟨*arg1*⟩ and gobbles ⟨*arg2*⟩.

```
\newcommand\tchklst@firstoftwoargs[3]{#1{#2}}
```

\tchklst@DisplayDeadline@i The \tchklst@DisplayDeadline@i{⟨*y*⟩}{⟨*m*⟩}{⟨*d*⟩}{⟨*deadline*⟩}{⟨*status*⟩} macro displays a given ⟨*deadline*⟩ that is additionally decomposed into year, month, and day given a particular ⟨*status*⟩ of the respective checklist entry. The macro uses the chosen output date format as well as colored highlighting.

```
\newcommand\tchklst@DisplayDeadline@i[5]{%
```

Check whether the entry is completed and whether the deadline has passed. Collect the checks' results as arguments for \CheckListHighlightDeadline.

```
  \def\tchklst@@args{}%
  \CheckListIfClosed{#5}%
    {\appto\tchklst@@args{{true}}}{\appto\tchklst@@args{{false}}}%
  \tchklst@ifafter{#1}{#2}{#3}
    {\appto\tchklst@@args{{true}}}{\appto\tchklst@@args{{false}}}%
```

Apply the selected output format for deadlines and apply the highlighting to the result.

```
  \expandafter\CheckListHighlightDeadline\tchklst@@args
    {\tchklst@@dateoutput@use{#1}{#2}{#3}{#4}}}
```

\CheckListHighlightDeadline The \CheckListHighlightDeadline{⟨*closed?*⟩}{⟨*passed?*⟩}⟨*deadline*⟩ macro highlights the given ⟨*deadline*⟩ based on the two Boolean ('true' or 'false') arguments ⟨*done?*⟩ (whether the respective checklist entry has been completed) and ⟨*passed?*⟩ (whether the deadline has passed). One can \renewcommand this macro to change the deadline highlighting.

```
\newcommand\CheckListHighlightDeadline[3]{%
  \ifbool{#1}
    {\textcolor{green!66!black}{#3}}
    {\ifbool{#2}{\textcolor{red}{#3}}
                {\textcolor{black}{#3}}}}
```

\tchklst@splitapply@i The following auxiliary macro just swaps the first two arguments, ⟨*text*⟩ and ⟨*sep*⟩ of \tchklst@splitapply such that the now first argument ⟨*sep*⟩ can be expanded more easily.

```
\newcommand*\tchklst@splitapply@i[2]{\tchklst@splitapply{#2}{#1}}
```

\tchklst@ifafter The \tchklst@ifafter{⟨*y*⟩}{⟨*m*⟩}{⟨*d*⟩}{⟨*iftrue*⟩}{⟨*iffalse*⟩} macro performs the check whether the current date is after the date specified by ⟨*y*⟩, ⟨*m*⟩, and ⟨*d*⟩. If this is the case, the macro expands to ⟨*iftrue*⟩, otherwise to ⟨*iffalse*⟩. Credits for this code go to http://tex.stackexchange.com/questions/41404/how-to-make-time-dependent-code!.

```
\newcommand*\tchklst@ifafter[3]{%
  \ifnum\the\year\two@digits\month\two@digits\day%
       >\numexpr#1\two@digits{#2}\two@digits{#3}\relax
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi}
```

\CheckListDateCompare The \CheckListDateCompare{⟨*date*⟩}{⟨*comp*⟩}{⟨*refdate*⟩}{⟨*iftrue*⟩}{⟨*iffalse*⟩}{⟨*iffail*⟩} macro compares ⟨*date*⟩ against ⟨*refdate*⟩ using the operator ⟨*comp*⟩. The latter must be one of <, =, and >. If the dates fulfill the comparison, the macro expands to ⟨*iftrue*⟩. If the dates do not fulfill the comparison, the macro expands to ⟨*iffalse*⟩. Finally, if one of ⟨*date*⟩ and ⟨*refdate*⟩ are not recognized as dates, the macro expands to ⟨*iffail*⟩.

```
\newcommand\CheckListDateCompare[6]{%
  \bgroup
  \def\do##1##2##3##4{\egroup
    \tchklst@DateCompare{#1}{#2}{##1}{##2}{##3}{#4}{#5}{#6}}%
  \CheckListParseDate{#3}{\do}{\egroup#6}}
```

\tchklst@DateCompare The \tchklst@DateCompare{⟨*date*⟩}{⟨*comp*⟩}{⟨*y*⟩}{⟨*m*⟩}{⟨*d*⟩}{⟨*iftrue*⟩}{⟨*iffalse*⟩}{⟨*iffail*⟩} macro is the internal counterpart to \CheckListDateCompare. The difference is that the former expects ⟨*refdate*⟩ to already be parsed into ⟨*y*⟩, ⟨*m*⟩, and ⟨*d*⟩.

```
\newcommand*\tchklst@DateCompare[8]{%
  \bgroup
  \def\do##1##2##3##4{\egroup
```

The actual comparsion between the dates is done via \ifnum. Here, the auxiliary macro \do{⟨*year*⟩}{⟨*month*⟩}{⟨*day*⟩}{⟨*date*⟩} gets the components of ⟨*date*⟩ from \CheckListParseDate.

```
    \ifnum##1\two@digits{##2}\two@digits{##3}%
      #2%
      #3\two@digits{#4}\two@digits{#5}\relax
    \expandafter\@firstoftwo
    \else\expandafter\@secondoftwo\fi{#6}{#7}}%
  \CheckListParseDate{#1}{\do}{\egroup#8}}
```

\CheckListParseDate The \CheckListParseDate{⟨*date*⟩}{⟨*cmd*⟩}{⟨*fail*⟩} macro parses ⟨*date*⟩ according to the selected date input format. If the parsing succeeds, the macro expands to

⟨*cmd*⟩{⟨*year*⟩}{⟨*month*⟩}{⟨*day*⟩}{⟨*date*⟩} (i.e., ⟨*cmd*⟩ must take four arguments). Otherwise, the macro expands to ⟨*fail*⟩{⟨*date*⟩} (i.e., ⟨*fail*⟩ must take one argument).

```
\newcommand\CheckListParseDate[3]{%
  \expandafter\tchklst@splitapply@i\expandafter{\tchklst@inputdate@sep}
    {#1}
```

Dates have three components and all must be positive numbers.

```
    {3}{\tchklst@ifPositive}
```

Before expanding to ⟨*cmd*⟩, reorder the parsed date components according to the ordering of the selected date input format.

```
    {\expandafter#2\tchklst@inputdate@order}
    {#3}
```

The following argument is applied to whichever of the two previous arguments `\tchklst@splitapply@i` expands to.

```
    {#1}}
```

The following set of macros is for registering date input and date output formats.

`\tchklst@@InputDateFormats`
`\tchklst@@OutputDateFormats`

The `\tchklst@@InputDateFormats` and `\tchklst@@OutputDateFormats` macros collect the list of known input date formats and, respectively, output date formats. Both are etoolbox lists. Initially, both lists are empty.

```
\newcommand*\tchklst@@InputDateFormats{}
\newcommand*\tchklst@@OutputDateFormats{}
```

`\tchklst@registerdateinputfmt`

The `\tchklst@registerdateinputfmt{`⟨*name*⟩`}{`⟨*reorder*⟩`}{`⟨*sep*⟩`}` macro registers a date input format under the given ⟨*name*⟩. The format uses ⟨*sep*⟩ as the separator symbol between dates' components and uses the ⟨*reorder*⟩ code to reorder the components from given dates to the ordering "⟨*year*⟩-⟨*month*⟩-⟨*day*⟩".

```
\newcommand*\tchklst@registerdateinputfmt[3]{%
  \listadd\tchklst@@InputDateFormats{#1}%
  \csgdef{tchklst@dateorder@#1}##1##2##3{#2}%
  \csgdef{tchklst@dateformat@sep@#1}{#3}}
```

The following registers some typical date input formats.

```
\tchklst@registerdateinputfmt{d.m.y}{{#3}{#2}{#1}}{.}
\tchklst@registerdateinputfmt{m/d/y}{{#2}{#3}{#1}}{/}
\tchklst@registerdateinputfmt{y-m-d}{{#1}{#2}{#3}}{-}
```

`\tchklst@registerdateoutputfmt`

The `\tchklst@registerdateoutputfmt{`⟨*name*⟩`}{`⟨*use*⟩`}` macro registers a date output format under the given ⟨*name*⟩. The ⟨*use*⟩ code may take four parameters: ⟨*year*⟩, ⟨*month*⟩, ⟨*day*⟩, and ⟨*deadline*⟩. Hence, ⟨*use*⟩ can use the original ⟨*deadline*⟩ as well as the decomposed form into year, month, and day.

```
\newcommand*\tchklst@registerdateoutputfmt[2]{%
  \listadd\tchklst@@OutputDateFormats{#1}%
  \csgdef{tchklst@dateoutput@use@#1}##1##2##3##4{#2}}
```

The following registers some typical date output formats.

```
\tchklst@registerdateoutputfmt{same}{#4}
\tchklst@registerdateoutputfmt{datetime}
```

```
  {\DTMdisplaydate{#1}{#2}{#3}{-1}}
\tchklst@registerdateoutputfmt{d.m.y}{#3.#2.#1}
\tchklst@registerdateoutputfmt{m/d/y}{#2/#3/#1}
\tchklst@registerdateoutputfmt{y-m-d}{#1-#2-#3}
\tchklst@registerdateoutputfmt{d.m.}{#3.#2.}
\tchklst@registerdateoutputfmt{m/d}{#2/#3}
\tchklst@registerdateoutputfmt{m-d}{#2-#3}
```

The following \numexprs looks a bit unfamiliar but it computes the modulo, given that integer division rounds the result.

```
\tchklst@registerdateoutputfmt{d.m.yy}
  {#3.#2.\the\numexpr #1-100*((#1-50)/100)\relax}
\tchklst@registerdateoutputfmt{m/d/yy}
  {#2/#3/\the\numexpr #1-100*((#1-50)/100)\relax}
\tchklst@registerdateoutputfmt{yy-m-d}
  {\the\numexpr #1-100*((#1-50)/100)\relax-#2-#3}
```

\tchklst@DateFailLax
\tchklst@DateFailStrict

The \tchklst@DateFailStrict{⟨*date*⟩} macro displays a failure to parse ⟨*date*⟩ in `strict-dates` mode. Conversely, \tchklst@DateFailLax{⟨*date*⟩} formats a failure to parse ⟨*date*⟩, in non-strict mode.

```
\newcommand\tchklst@DateFailStrict[1]{%
  \PackageError{typed-checklist}
    {date `#1' not understood}
    {See the options `strict-dates' and `input-dates'
    in the package documentation\MessageBreak
    if you intend to keep the value `#1'.}}
\newcommand\tchklst@DateFailLax[1]{\textit{#1}}
```

The remainder of this section defines generic auxiliary macros for deadline parsing.

\tchklst@splitapply

The \tchklst@splitapply{⟨*text*⟩}{⟨*sep*⟩}{⟨*n*⟩}{⟨*cond*⟩}{⟨*cmd*⟩}{⟨*fail*⟩} macro is a generic macro for parsing a list-like ⟨*text*⟩ of fixed length ⟨*n*⟩, whose components satisfy ⟨*cond*⟩ (a macro with one argument) and are separated by ⟨*sep*⟩. If c_1 to c_n are the components of ⟨*text*⟩, i.e., if ⟨*text*⟩=c_1⟨*sep*⟩$\cdots$⟨*sep*⟩c_n, then the macro expands to ⟨*cmd*⟩{⟨c_1⟩}$\cdots${⟨c_n⟩}. If ⟨*text*⟩ has less than ⟨*n*⟩ or more than ⟨*n*⟩ components, or if at least one of the components does not satisfy ⟨*cond*⟩, then the macro expands to ⟨*fail*⟩.

```
\newcommand*\tchklst@splitapply[6]{%
```

The \tchklst@split@@rec{⟨*k*⟩}{⟨*cmd*⟩}{⟨*prefix*⟩}⟨*sep*⟩{⟨*suffix*⟩}\relax macro recursively grabs ⟨*prefix*⟩es from the remaining ⟨*suffix*⟩ and appends them to ⟨*cmd*⟩. The latter accumulates ⟨*cmd*⟩ and the already parsed components.

```
  \def\tchklst@split@@rec##1##2##3#2##4\relax{%
```

Check whether ⟨*cond*⟩ holds for ⟨*prefix*⟩ first.

```
    #4{##3}
```

If ⟨*k*⟩ > 0, i.e., then ⟨*suffix*⟩ contains the last ⟨*k*⟩ components of ⟨*text*⟩ plus a trailing ⟨*sep*⟩.

```
      {\ifnumgreater{##1}{0}%
```

If ⟨*suffix*⟩ is empty, then ⟨*text*⟩ contained too few components and, hence, expand to ⟨*fail*⟩. Otherwise recurse.

```
        {\ifstrempty{##4}
          {#6}
          {\tchklst@split@@rec{##1-1}{##2{##3}}##4\relax}}%
```

Otherwise, if ⟨k⟩ = 0, and ⟨*suffix*⟩ is empty, then ⟨*text*⟩ indeed contains ⟨n⟩ components and ⟨*prefix*⟩ is appended to ⟨*cmd*⟩ as the last component. If ⟨*suffix*⟩ is nonempty, expand to ⟨*fail*⟩.

```
        {\ifstrempty{##4}
          {##2{##3}}
          {#6}}}
```

If ⟨*cond*⟩ does not hold, expand to ⟨*fail*⟩.

```
      {#6}}%
  \tchklst@split@@rec{#3-1}{#5}#1#2\relax}
```

`\tchklst@ifPositive` The `\tchklst@ifPositive{`⟨*text*⟩`}{`⟨*iftrue*⟩`}{`⟨*iffalse*⟩`}` macro expands to ⟨*iftrue*⟩ if ⟨*text*⟩ is a positive number and expands to ⟨*iffalse*⟩ otherwise (i.e., if ⟨*text*⟩ is not a number or not positive). The code of the macro is taken from Donald Arseneau's cite package.

```
\newcommand*\tchklst@ifPositive[1]{%
  \ifcat _\ifnum\z@<0#1_\else A\fi
    \expandafter\@firstoftwo \else \expandafter\@secondoftwo \fi}
```

10.10 Default Checklist Types and States

We use some packages for the default symbols in the checklist.

```
\RequirePackage{bbding}
```

The following line makes sure that the bbding font is actually loaded, by simply putting a particular symbol into a box and then forgetting the box again (via the grouping). This addresses the case that the bbding symbols are used inside an `\import*` or `\subimport*` of the import package: In this case, the font would be attempted to be loaded only inside the 'import' and could then no longer be found (producing "No file Uding.fd").

```
\AtBeginDocument{{\setbox0\hbox{\Checkmark}}}
```

The following provides the default set of checklist types.

```
\CheckListAddType{Goal}{$\bigcirc$}
\CheckListAddType{Task}{{\small\Square}}
\CheckListAddType{Artifact}{{\large$\bigtriangleup$}}
\CheckListAddType{Milestone}{\FiveStarOpen}
```

The following provides the default set of status possibilities.

```
\CheckListAddStatus{Goal,Task,Milestone}{open}{false}{}
\CheckListAddStatus{Goal}{dropped}{true}{\tiny\XSolid}
\CheckListAddStatus{Task}{dropped}{true}{\small\XSolid}
\CheckListAddStatus{Goal}{unclear}{false}{\footnotesize ?}
\CheckListAddStatus{Task}{unclear}{false}%
                    {\raisebox{0.4ex}{\hbox{\footnotesize ?}}}
\CheckListAddStatus{Artifact}{unclear}{false}%
```

```
                    {\raisebox{0.3ex}{\hbox{\tiny\bfseries ?}}}

\CheckListAddStatus{Goal}{achieved}{true}{\kern 4pt\Checkmark}
\CheckListAddStatus{Milestone}{achieved}{true}{\FiveStar}

\CheckListAddStatus{Task}{started}{false}%
                    {\kern 1pt\small\ArrowBoldRightStrobe}
\CheckListAddStatus{Task}{done}{true}{\kern 2pt\Checkmark}

\CheckListAddStatus{Artifact}{missing}{false}{}
\CheckListAddStatus{Artifact}{incomplete}{false}%
                    {\kern 1pt{\tiny\ArrowBoldRightStrobe}}
\CheckListAddStatus{Artifact}{available}{true}{\kern 4pt\Checkmark}
\CheckListAddStatus{Artifact}{dropped}{true}{{\small$\dagger$}}
```

10.11 Default Checklist Layouts

The following provides the default set of checklist layouts.

10.11.1 list

We use the marginnote package to display deadlines in the `list` layout.

```
\RequirePackage{marginnote}
```

The `list` layout is based on a `description` environment with a slightly modified vertical and horizontal spacing.

```
\CheckListDeclareLayout{list}{status,label,description,
                              who,deadline+status,END}%
  {\bgroup\topsep=\medskipamount\itemsep=0pt\itemize\@newlistfalse}%
  {\global\@newlistfalse\enditemize\egroup}
```

The checklist entry starts with the status symbol, which opens up a new list item.

```
\CheckListDefineFieldFormat{list}{status}%
  {\item[{\normalfont\CheckListStatusSymbol{#1}}]}
```

Show the label in the reverse margin, with some nice layout.

```
\CheckListDefineFieldFormat{list}{label}{%
  \ifstrempty{#1}{}{%
    \CheckListDefaultLabel{#1}%
    \ifbool{inner}%
      {\mbox{\small(\ref{#1})}%
        \nobreak\hskip 0pt plus50pt\allowbreak
        \ \hskip 0pt plus-50pt\relax}%
      {\leavevmode\reversemarginpar\marginpar{%
       \textcolor{gray}{\underbar{\hbox to \hsize{%
       \normalfont\textcolor{black}{\ref{#1}}\hfil}}}}}}
```

Show the description, with leading spaces removed.

```
\CheckListDefineFieldFormat{list}{description}{%
    \ignorespaces #1\relax}
```

Show the responsible person(s), if the `who` option is given in ⟨*options*⟩.

```
\CheckListDefineFieldFormat{list}{who}{%
  \CheckListSigned[#1]{\textit{(#1)}}}
```

Show the deadline of the entry in the margin, if the `deadline` option is given in ⟨*options*⟩.

```
\CheckListDefineFieldFormat{list}{deadline+status}{%
  \ifstrempty{#1}{}{{\normalmarginpar\marginnote{%
```

The following `\unskip` prevents `\marginnote` from breaking an overfull margin text at it's very beginning, which meant that the margin text would vertically be placed below the actual entry (see also https://tex.stackexchange.com/questions/117695/).

```
    \unskip
    \CheckListDisplayDeadline{#2}{#1}}}}}
```

End the display of one checklist entry. ⟨*options*⟩.

```
\CheckListDefineFieldFormat{list}{END}{{%
  \parfillskip=0pt \finalhyphendemerits=0 \endgraf}}
```

10.11.2 hidden

The `hidden` layout completely hides the checklist and all its entries. We add the `status` field only to ignore spaces after each entry.

```
\CheckListDeclareLayout{hidden}{dummy}{\ignorespaces}{\ignorespaces}
\CheckListDefineFieldFormat{hidden}{dummy}{\ignorespaces}
```

10.11.3 table

The `table` layout formats the checklist as a table, one row per checklist entry. The `NC` field just inserts the column separator.

```
\CheckListDeclareLayout{table}%
  {newline,status,NC,label,description,NC,who,NC,deadline+status}%
  {%
    \tchklst@@begintab\hline
```

The `\tchklst@@newline` macro ensures that the `\\\hline` is only produced after the first content row. Generally, `\tchklst@@newline` (and the `newline` field) are used at the *beginning* of each row is due to checklist entry filters: This way of line breaking ensures there is no spurious empty row at the end of tables whose last checklist entry was filtered out.

```
    \gdef\tchklst@@newline{\\\hline\tchklst@@endhead
      \gdef\tchklst@@newline{\\\hline}}%
    \textbf{Status} & \textbf{Description} &
    \textbf{Who} & \textbf{Deadline}}
  {\tchklst@@endtab}
\CheckListDefineFieldFormat{table}{newline}{\tchklst@@newline}
\CheckListDefineFieldFormat{table}{status}{\CheckListStatusSymbol{#1}}
\CheckListDefineFieldFormat{table}{label}%
  {\ifstrempty{#1}{}{%
     \leavevmode\CheckListDefaultLabel{#1}%
     \mbox{\small(\ref{#1})}%
     \nobreak\hskip 0pt plus50pt\allowbreak
     \ \hskip 0pt plus-50pt\relax}}
\CheckListDefineFieldFormat{table}{description}{\ignorespaces #1}
```

```
\CheckListDefineFieldFormat{table}{deadline+status}{%
  \ifstrempty{#1}{}{\CheckListDisplayDeadline{#2}{#1}}}
\CheckListDefineFieldFormat{table}{who}{#1}
\CheckListDefineFieldFormat{table}{NC}{&}
```

The following macros define the package-specific table code.

\tchklst@inittab@xltabular
\tchklst@begintab@xltabular
\tchklst@endtab@xltabular

The following three macros specify how the xltabular package is initialized (i.e., how the package is loaded) and how table environments are started and, respectively, ended.

```
\newcommand\tchklst@inittab@xltabular{%
  \RequirePackage{array,xltabular}}
\newcommand\tchklst@begintab@xltabular{%
  \setlength{\extrarowheight}{0.5ex}%
  \def\tchklst@@endhead{\endhead}%
```

The following modifies the internal end code of xltabular such that `\endxltabular` can be the first token as required while there is still the horizontal line.

```
  \preto\XLT@ii@TX@endtabularx{%
    \toks@\expandafter{\the\toks@\tchklst@@newline}}%
  \xltabular{\linewidth}{|c|X|l|r|}}
\newcommand\tchklst@endtab@xltabular{\endxltabular}
```

\tchklst@inittab@tabularx
\tchklst@begintab@tabularx
\tchklst@endtab@tabularx

The following three macros specify how the tabularx package is initialized (i.e., how the package is loaded) and how table environments are started and, respectively, ended.

```
\newcommand\tchklst@inittab@tabularx{%
  \RequirePackage{array,tabularx}}
\newcommand\tchklst@begintab@tabularx{%
  \let\tchklst@@endhead\relax%
  \setlength{\extrarowheight}{0.5ex}%
```

The following is analogous to its counterpart in `xltabular`.

```
  \preto\TX@endtabularx{\toks@\expandafter{\the\toks@\tchklst@@newline}}%
  \tabularx{\linewidth}{|c|X|l|r|}}
\newcommand\tchklst@endtab@tabularx{\endtabularx}
```

\tchklst@inittab@ltablex
\tchklst@begintab@ltablex
\tchklst@endtab@ltablex

The following three macros specify how the ltablex package is initialized (i.e., how the package is loaded) and how table environments are started and, respectively, ended.

```
\newcommand\tchklst@inittab@ltablex{\RequirePackage{ltablex}
```

The following fixes a bug in ltablex, see https://tex.stackexchange.com/a/197000/132738.

```
  \patchcmd{\TX@endtabularx}
    {\end{tabularx}}
    {\endtabularx\endgroup}
    {}
    {\PackageError{typed-checklist}{Could not apply code patch to
       ltablex' package.}{}}}
\let\tchklst@begintab@ltablex=\tchklst@begintab@tabularx
\let\tchklst@endtab@ltablex=\tchklst@endtab@tabularx
```

10.12 Package Options

10.12.1 Package-Only Options

The `withAsciilist` option enables support for the asciilist package.

```
\define@boolkey[tchklst]{PackageOptions}[tchklst@]
  {withAsciilist}[true]{}
```

The `tablepkg` option specifies which table package is used for layouting checklists in the `table` layout.

```
\define@choicekey[tchklst]{PackageOptions}{tablepkg}[\val]
                  {ltablex,tabularx,xltabular}{%
  \letcs\tchklst@@inittab{tchklst@inittab@\val}%
  \letcs\tchklst@@begintab{tchklst@begintab@\val}%
  \letcs\tchklst@@endtab{tchklst@endtab@\val}%
}
```

The `onecounter` option specifies whether a single counter shall be used for all entry labels, no matter the entry types, or whether one counter per entry type shall be used.

```
\define@boolkey[tchklst]{PackageOptions}[tchklst@]
  {onecounter}[true]{}
```

10.12.2 Processing Options

Set option defaults and then load the given options.

```
\ExecuteOptionsX[tchklst]<PackageOptions,GlobalListOptions>{%
  withAsciilist=false,
  tablepkg=xltabular,
  onecounter=true,
  layout=list,
  input-dates=d.m.y,
  output-dates=same,
  strict-dates=false,
}
\ProcessOptionsX[tchklst]<PackageOptions,GlobalListOptions>\relax
\tchklst@@inittab
```

10.12.3 Labels

`\CheckListDefaultLabel` The `\CheckListDefaultLabel{`⟨*label*⟩`}` macro puts the given ⟨*label*⟩ and ensures that this ⟨*label*⟩ is based on the right checklist entry counter.

```
\newcommand*\CheckListDefaultLabel[1]{%
  \ifstrempty{#1}{}
    {\ifbool{tchklst@onecounter}
      {\refstepcounter{tchklst@entryID}}
      {\refstepcounter{tchklst@entryID@\tchklst@cur@type}}%
    \label{#1}}}
```

`\tchklst@NewEntryCounter` The `\tchklst@NewEntryCounter[`⟨*at*⟩`]{`⟨*type*⟩`}` creates a new counter for checklist entries and defines the format for displaying counter values. The counter is named `tchklst@entryID`⟨*at*⟩⟨*type*⟩.

```
\newcommand*\tchklst@NewEntryCounter[2][@]{%
  \newcounter{tchklst@entryID#1#2}%
  \setcounter{tchklst@entryID#1#2}{0}%
  \ifstrempty{#2}
    {\csgdef{thetchklst@entryID#1#2}{%
       \tchklst@cur@type~\protect\textsc{\roman{tchklst@entryID#1#2}}}}%
    {\csgdef{thetchklst@entryID#1#2}{%
       #2~\protect\textsc{\roman{tchklst@entryID#1#2}}}}}
```

If the package shall use a single counter for all entries then define the counter and how counter values are displayed here.

```
\iftchklst@onecounter
  \tchklst@NewEntryCounter[]{}
\else
```

Otherwise, register the creation of a new type-specific counter in the hook for new checklist types.

```
  \tchklst@IntroduceTypeHook{\tchklst@NewEntryCounter[@]}
\fi
```

10.12.4 asciilist

If the package is loaded with asciilist support...

```
\iftchklst@withAsciilist
```

First, we load the package.

```
\RequirePackage{asciilist}
```

`\tchklst@ChkListEntry` The `\tchklst@ChkListEntry{`⟨*item-macro*⟩`}{`⟨*content*⟩`}` macro can be used as a parameter to `\AsciiListEndArg` of the asciilist package in order to allow for checklist entries in an `AsciiList`.

```
\newcommand*\tchklst@ChkListEntry[2]{%
  \tchklst@ChkListEntry@i{#1}#2\@undefined}
```

The used auxiliary macros serve the purpose of parsing the input and have the following signatures:

- `\tchklst@ChkListEntry@i{`⟨*item-macro*⟩`}{`⟨*status+opts*⟩`}{`⟨*descr*⟩`}`
- `\tchklst@ChkListEntry@ii{`⟨*item-macro*⟩`}{`⟨*descr*⟩`}{`⟨*status*⟩`}{`⟨*opts*⟩`}`

```
\def\tchklst@ChkListEntry@ii#1#2#3[#4]#5\@undefined{#1[#4]{#3}{#2}}
\def\tchklst@ChkListEntry@i#1#2:#3\@undefined{%
  \tchklst@ChkListEntry@ii{#1}{#3}#2[]\@undefined}
```

`\tchklst@RegisterAsciiTypeEnv` The `\tchklst@RegisterAsciiTypeEnv{`⟨*type*⟩`}` registers an asciilist environment for the given checklist ⟨*type*⟩.

```
\newcommand*\tchklst@RegisterAsciiTypeEnv[1]{%
  \AsciiListRegisterEnv{#1List}%
    {\tchklst@aux@OargAfter{\CheckList{#1}}}%
    {\endCheckList}%
    {\AsciiListEndArg{\tchklst@ChkListEntry{\csname #1\endcsname}}}}%
\tchklst@IntroduceTypeHook{\tchklst@RegisterAsciiTypeEnv}
\fi
```

Change History

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

P

R

S

T

U

V

W

X

Y

Z