The typed-checklist package*

Richard Grewe
r-g+tex@posteo.net

October 31, 2018

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 v1.5b, dated 2018/10/31. 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 persons 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 ones 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.

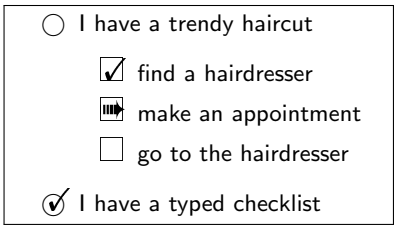
7. milestones in milestones . . . . . ☠

   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.

`\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. The following keys can be set:

- With the `layout` key, the layout of the checklist can be chosen. Supported layouts are `list` (a list-based layout with each entry being a list item), `table` (a table-based layout with each entry being a table row), and `hidden` (a layout that does not display the entries).

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.

`\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.

`deadline` This option declares a deadline for the checklist entry, i.e., a date until which the entry must be addressed at latest. The deadline must be of the format "⟨*day*⟩.⟨*month*⟩.⟨*year*⟩".

[1] Incomplete sentences typically tend to be less clear.
[2] See Section 4.2 to find out how custom states can be defined
[3] See Section 4.3 to find out how custom ⟨*key*⟩s can be defined.

`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[4]. 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.

The ⟨*options*⟩ parameter can be set as documented for the `\Goal` macro on page 4.

`\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.

[4] For a ⟨*description*⟩ in imperative form, the `who` option can be used to specify who is addressed by the ⟨*description*⟩.

`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.

3.1 Example

The following example shows the use of nested checklists and the use of the options `layout`, `deadline`, `label`, and `who`. Note that deadlines are displayed in the margin of the document, which in this documentation is outside the example box in this documentation.

☑ No Y1K problems 31.12.999

◯ No Y2K problems . . . . . *(John)* 31.12.1999

Status	Description	Who	Deadline
▶	(Task 1) Repair all programs	John	
☐	Just turn off all computers, if Task 1 fails	Mankind	31.12.1999

? No Y10K problems 31.12.9999

```
\begin{CheckList}{Goal}
  \Goal[deadline=31.12.999]{achieved}{No Y1K problems}
  \Goal[who=John,deadline=31.12.1999]{open}{No Y2K problems}
    \begin{CheckList}[layout=table]{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=31.12.9999]{unclear}{No Y10K problems}
\end{CheckList}
```

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

`\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 4.2) 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.

4.2 Defining Checklist Entry States

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

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.

4.3 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 that can be the names of the checklist entry options (plus "description" and "status") but can assume other values.

`\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 argument, through which it is passed the entry's option with name ⟨*field*⟩.

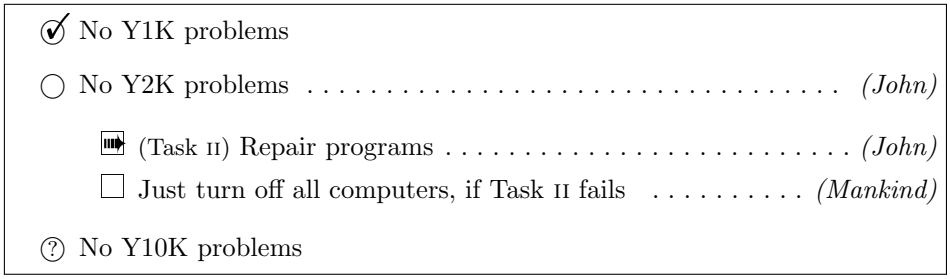
`\CheckListExtendLayout{`⟨*name*⟩`}{`⟨*base*⟩`}{`⟨*fields*⟩`}`

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`.

5 Checklists and Other Packages

5.1 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 3.1:




```
\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 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 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 tood package and shares the main differences to the typed-checklist package.

todonotes:
The package is similar in spirit to todo and easy-tood, but provide 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).

7 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.

- 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 date format for deadlines currently is "DD.MM.YYYY". We could make the package more flexible in this regard by offering other formats as well, for instance by using the datetime2 package.

- 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 inspired by the following stackexchange thread: http://tex.stackexchange.com/questions/6988/how-to-sort-an-alphanumeric-list

8 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

9 Implementation

9.1 Package Options

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

```
\RequirePackage{xkeyval}
```

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

```
\define@boolkey{typed-checklist.sty}[tchklst@]{withAsciilist}{}
\ProcessOptionsX
```

9.2 Basic Package Dependencies

We use the etoolbox package for simpler handling of lists.

```
\RequirePackage{etoolbox}
```

We use colors for deadlines, for instance.

```
\RequirePackage{xcolor}
```

If the package is loaded with asciilist support, we load the package here.

```
\iftchklst@withAsciilist
\RequirePackage{asciilist}
\fi
```

9.3 Checklist and Entry Options

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

```
\define@cmdkey[tchklst]{ListOption}{layout}[\tchklst@defaultlayout]{}
\presetkeys[tchklst]{ListOption}{layout}{}
```

`\CheckListDefaultLayout` The `\CheckListDefaultLayout{`⟨*layout*⟩`}` macro sets the default layout for all `CheckList` environments that do not set the `layout` option explicitly.

```
\newcommand*\CheckListDefaultLayout[1]{%
  \ifinlist{#1}{\tchklst@ChecklistLayouts}{}{%
    \PackageError{typed-checklist}{%
      Checklist layout '#1' cannot be made default:
      it does not exist}{}}%
  \def\tchklst@defaultlayout{#1}}
\def\tchklst@defaultlayout{list}
```

`\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]{EntryOption}{#1}[#2]{}%
  \presetkeys[tchklst]{EntryOption}{#1}{}}
```

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

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

9.4 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.

```
  \expandafter\def\csname tchklst@ChecklistTypeSym@#1\endcsname{#2}%
```

Create an initially empty list of possible states that entries of the type can have.

```
  \expandafter\def\csname tchklst@ChecklistStates@#1\endcsname{}%
```

If asciilist support is enabled, register an environment for the checklist type.

```
  \iftchklst@withAsciilist
    \AsciiListRegisterEnv{#1List}%
      {\tchklst@aux@OargAfter{\CheckList{#1}}}%
      {\endCheckList}%
      {\AsciiListEndArg{\tchklst@ChkListEntry{\csname #1\endcsname}}}%
  \fi
}
```

`\tchklst@aux@OargAfter` The `\tchklst@aux@OargAfter{`⟨*macro-use*⟩`}[`⟨*opt-arg*⟩`]` macro takes a ⟨*macro-use*⟩ without optional arguments and subsequently an optional argument. If the optional argument is given, then this is added to the ⟨*macro-use*⟩ in a way that the macro uses the optional argument. If no optional argument is given, then the ⟨*macro-use*⟩ is taken as is.

Example use: `\tchklst@aux@OargAfter{\cite{foo}}[page 9]` would expand to `\tchklst@aux@OargAfter@ii{\cite}{page 9}{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}}}}
```

9.5 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}}}}
```

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

```
\newcommand*\tchklst@getsymbol[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}%
  \rlap{\hbox to \wd0{\hss #1\hss}}\box0 }}
```

`\tchklst@ifsymdone` The `\tchklst@ifsymdone{`⟨*type*⟩`}{`⟨*status*⟩`}{`⟨*iftrue*⟩`}{`⟨*iffalse*⟩`}` macro expands to ⟨*iftrue*⟩, if the ⟨*status*⟩ of an entry in a checklist of type ⟨*type*⟩ is a "closed" one (see the documentation for `\CheckListAddStatus` for details). Otherwise, the macro expands to ⟨*iffalse*⟩.

```
\newcommand*\tchklst@ifsymdone[2]{%
  \csname if\csname tchklst@isclosed@#1@#2\endcsname\endcsname
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi}
```

9.6 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*⟩ in a checklist of the given ⟨*layout*⟩. The code may take one argument (i.e., use `#1`).

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

`\tchklst@FormattedField` The `\tchklst@FormattedField{`⟨*field*⟩`}` macro returns the macro for formatting the given ⟨*field*⟩ in a layout given by `\tchklst@cur@layout`.

```
\newcommand*\tchklst@FormattedField[1]{%
  \csname tchklst@ChecklistFormat@\tchklst@cur@layout @#1\endcsname}
```

`\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}}}}
```

9.7 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]{ListOption}{#1}%
  \tchklst@CheckLayout{\cmdtchklst@ListOption@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=\cmdtchklst@ListOption@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@toks` The `\tchklst@entry@toks` token register is used by `\tchklst@entry` to first collect all the fields before showing the result. This is useful in cases when the layout changes the LaTeX grouping between the field display (as it is the case for a table layout).

```
\newtoks\tchklst@entry@toks
```

`\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 9.3 for the list of available options.

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

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]{EntryOption}{#1}%
```

Define the label of the entry, if the `label` option is given in ⟨*options*⟩.

```
\ifx\cmdtchklst@EntryOption@label\empty\else
  \refstepcounter{tchklst@entryID}%
  \expandafter\label\expandafter{\cmdtchklst@EntryOption@label}%
\fi
```

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

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

Show the fields of the entry in the order they were given.

```
\tchklst@entry@toks={}%
\def\do##1{%
  \begingroup
  \edef\tchklst@doformat{\endgroup
    \noexpand\tchklst@entry@toks={%
      \expandonce{\the\tchklst@entry@toks}%
      \noexpand\tchklst@FormattedField{##1}%
      {\csexpandonce{cmdtchklst@EntryOption@##1}}}}%
  \tchklst@doformat}%
\dolistloop\tchklst@cur@fields
\the\tchklst@entry@toks}
```

`\tchklst@ifafterdots` The `\tchklst@ifafterdots` ⟨*day*⟩.⟨*month*⟩.⟨*year*⟩ macro is a parsing macro for dates in dotted notation. The macro is a wrapper for `\tchklst@ifafter`.

```
\def\tchklst@ifafterdots #1.#2.#3\relax{\tchklst@ifafter{#1}{#2}{#3}}
```

`\tchklst@ifafter` The `\tchklst@ifafter{`⟨*day*⟩`}{`⟨*month*⟩`}{`⟨*year*⟩`}{`⟨*iftrue*⟩`}{`⟨*iffalse*⟩`}` macro performs the check whether the current date is after the date specified by ⟨*day*⟩, ⟨*month*⟩, and ⟨*year*⟩. 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#3\two@digits{#2}\two@digits{#1}\relax
    \expandafter\@firstoftwo
  \else
    \expandafter\@secondoftwo
  \fi}
```

`\tchklst@signed` The `\tchklst@signed{`⟨*text*⟩`}` macro is taken from Knuth's TEXbook with minor spacing modifications. See also http://tex.stackexchange.com/a/13761.

```
\def\tchklst@signed #1{{%
  \leavevmode\unskip\nobreak\hfil\penalty50\hskip0.25em
  \hbox{}\nobreak\dotfill\hbox{#1}}}
```

tchklst@entryID We define a counter for the labels of checklist entries. We also determine how counter values are displayed.

```
\newcounter{tchklst@entryID}
\setcounter{tchklst@entryID}{0}
\renewcommand*\thetchklst@entryID{%
  \tchklst@cur@type~\protect\textsc{\roman{tchklst@entryID}}}
```

9.8 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$\dagger$}}
```

9.9 Default Checklist Layouts

The following provides the default set of checklist layouts.

9.9.1 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,END}%
  {\bgroup\topsep=\medskipamount\itemsep=0pt\itemize}%
  {\enditemize\egroup}
```

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

```
\CheckListDefineFieldFormat{list}{status}%
  {\item[{\normalfont\tchklst@getsymbol{#1}}]}
```

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

```
\CheckListDefineFieldFormat{list}{label}{%
  \ifstrempty{#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}{%
  \ifstrempty{#1}{\nobreak\hfill\null}{%
    \tchklst@signed{\textit{(#1)}}}}
```

Show the deadline of the entry in the margin, if the `deadline` option is given in ⟨*options*⟩. **FIXME:** here, the interface of the code is not very elegant, because the field format code uses the internal macro `\cmdtchklst@EntryOption@status` for obtaining the current entry's status.

```
\CheckListDefineFieldFormat{list}{deadline}{%
  \ifstrempty{#1}{}{{\normalmarginpar\marginnote{%
     \tchklst@DisplayDeadline{\cmdtchklst@EntryOption@status}{#1}}}}}
```

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

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

`\tchklst@DisplayDeadline` The `\tchklst@DisplayDeadline{`⟨*status*⟩`}{`⟨*deadline*⟩`}` formats a ⟨*deadline*⟩ dependent on the ⟨*status*⟩ and the current date.

```
\newcommand\tchklst@DisplayDeadline[2]{%
```

Check which text color to use for this item if its deadline has already passed.

```
  \tchklst@ifsymdone{\tchklst@cur@type}{#1}%
    {\def\tchklst@deadcolor{green!66!black}}%
    {\def\tchklst@deadcolor{red}}%
```

Check whether the deadline of the entry has already passed and, if so, set the text color to the color determined above.

```
  \tchklst@ifafterdots#2\relax%
    {\textcolor{\tchklst@deadcolor}}%
    {}%
```

Show the actual deadline. Note that this may constitute the second parameter of the above `\textcolor`.

```
    {#2}}
```

9.9.2 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}
```

9.9.3 table

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

```
\RequirePackage{longtable,tabu}
\CheckListDeclareLayout{table}%
  {status,NC,label,description,NC,who,NC,deadline,endline}%
  {%
    \tabulinesep=0.5ex
    \longtabu to \linewidth {|c|X|r|r|}
    \hline
    \bf Status & \bf Description & \bf Who & \bf Deadline\endhead\hline}
  {\endlongtabu}
\CheckListDefineFieldFormat{table}{status}{\tchklst@getsymbol{#1}}
\CheckListDefineFieldFormat{table}{label}%
  {\ifstrempty{#1}{}{\mbox{\small(\ref{#1})}%
     \nobreak\hskip 0pt plus50pt\allowbreak
     \ \hskip 0pt plus-50pt\relax}}
\CheckListDefineFieldFormat{table}{description}{\ignorespaces #1}
\CheckListDefineFieldFormat{table}{deadline}{#1}
\CheckListDefineFieldFormat{table}{who}{#1}
\CheckListDefineFieldFormat{table}{NC}{&}
\CheckListDefineFieldFormat{table}{endline}{\\\hline}
```

9.10 Compatibility with Other Packages

9.10.1 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.

```
\iftchklst@withAsciilist
\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@CheckListEntry@i{⟨*item-macro*⟩}{⟨*status+opts*⟩}{⟨*entry*⟩} where ⟨*entry*⟩ is the goal/task/... of the checklist entry.
- \tchklst@CheckListEntry@ii{⟨*item-macro*⟩}{⟨*entry*⟩}{⟨*status*⟩}{⟨*options*⟩}.

```
\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}
\fi
```

Change History

Index

D

E

F

H

I

K

L

M