{cprotect.sty}
\verbatim in \macro arguments*

Bruno Le Floch†

Released 2010/12/30

Contents

1 Include \verb anywhere!

The cprotect package will allow you to put verbatim in footnotes[1] in a straightforward way. The section above was typeset using

```
\cprotect\section{Include\verb-\verb- anywhere!}
```

and the footnote was

```
...tes\cprotect\footnote{Like this: \verb-!@#$%^&*()_+-.}
```

*This file describes version v1.0, last revised 2010/12/30.

†E-mail: bruno@le-floch.fr

[1]Like this: !@#$%^&*()_+.

More generally, let us assume that you want to use verbatim text in the argument of some macro `\cs {`⟨*arg1*⟩`}`, and that this macro is normally allergic to verbatim. It probably has a good reason to be allergic, but using an auxiliary file, we can solve the problem entirely: if you want `{`⟨*arg1*⟩`}` to contain verbatim, just write `\cprotect\cs` or `\cprotect{\cs}` instead of `\cs`.[2] All the examples I give use very standard macros, but it should work with any macro.[3]

Braces are useful if a macro has *several* arguments, and you want to put verbatim in the second one: then type `\cprotect{\cs{`⟨*arg1*⟩`}}{`⟨*arg2*⟩`}`. If you want to put verbatim in both arguments, well ... it is not implemented yet, but if you send me an email saying why you need it, I will look into it (suggestions are, as always, welcome).

2 List of user commands

`\cprotect` Possibly the single most useful command is `\cprotect`, used as `\cprotect \foo {`⟨*arg1*⟩`}` or `\cprotect {`⟨*cslist*⟩`} {`⟨*arg1*⟩`}`. As described in the previous section, the first form behaves as `\foo {`⟨*arg1*⟩`}`, and the second as ⟨*cslist*⟩ `{`⟨*arg1*⟩`}`. The difference is that the argument `{`⟨*arg1*⟩`}` can now contain `\catcode` changes (e.g., induced by the `\verb` command and the `verbatim` environment). In fact, `{`⟨*arg1*⟩`}` is written to a file, and then read again as the argument of `\foo`. So using `\cprotect`, one could in principle build weird macros that read their arguments several times, with different `\catcodes` in effect each time. The macro `\ReadVerbatimUntil` below gives a better way of doing that.

`\cMakeRobust` If you find yourself using a `\cprotect\foo` combination frequently, you can simply define `\bar` to mean `\cprotect\foo`. It should work at least most of the time. You can also use the typical

```
\let\oldfoo\foo
\def\foo{\cprotect\oldfoo}
```

to redefine `\foo` itself instead of a new command `\bar`. The package provides a wrapper for this construction: `\cMakeRobust\foo` replaces `\foo` by a version which accepts verbatim. As a stupid example,

[2] This solves most problems, for instance, verbatim text in section titles written as `\cprotect\section{`⟨*title*⟩`}` appears correctly in the table of contents. However, there are still bugs in capitalized headers.

[3] If you have a macro that works when it is not preceeded by `\cprotect`, but breaks down if you put `\cprotect` in front of it, I will be interested to know why.

```
\newcommand{\expon}[1]{\mathrm{#1}^{#1}}
\cMakeRobust{\expon}
\(\expon{Hel\verb+|+o}\)
```

produces Hel|o$^{Hel|o}$, and the verbatim is treated correctly.

\cprotEnv Something similar to \cprotect exists for environments: \cprotEnv. Simply put \cprotEnv \body {⟨*name*⟩} instead of \body {⟨*name*⟩}. For example,

```
\cprotEnv\begin{align}
x&=\begin{cases}
  1 &\text{if }\verb"@#%^&*"\\
  2 &\text{otherwise}
\end{cases}
\end{align}
```

gives

$$x = \begin{cases} 1 & \text{if } \texttt{@\#\%\^{}\&*} \\ 2 & \text{otherwise} \end{cases} \tag{1}$$

Note that unfortunately we need to put the \verb outside \text. This is because currently, nesting of \cprotect and friends is not supported.

You can use \CPTbegin as a short-hand for \cprotEnv\begin, but this will fail when nesting the same environment twice: the inner nesting is distinguished by *precisely* the characters \ b e g i n { *n a m e* }.

\ReadVerbatimUntil Finally, the most powerful and the root of all these macros, from which the package currently derives its name, is \ReadVerbatimUntil. The aim of this command is to read a piece of text verbatim, until it reaches an end-marker. It then writes all that it has read to an auxiliary file for future use. The most naive approach has a major flaw when nesting is involved: when parsing {}} for instance, with an end-marker of }, we often wish to stop at the *second* closing bracket, not the first one. Thus, \ReadVerbatimUntil needs to be aware of a begin-marker. Also, for some applications we need to write things before and after the content read by \ReadVerbatimUntil. Finally, we want to do something once the file has been written, and possibly something before.

The syntax is thus \ReadVerbatimUntil [⟨*arg1*⟩] {⟨*arg2*⟩} ^begintext^endtext^begintag^endtag^, followed by the text in which we look for endtag. The caret (^) can be any character. It delimits the four verbatim-like arguments of \ReadVerbatimUntil. The strings of letters begintext

and `endtext` are pre- and ap-pended to the file. As mentionned before, `begintag` and `endtag` are used when reading the content, to determine how far `\ReadVerbatimUntil` should go. The mandatory argument `{`⟨*arg2*⟩`}` is executed once the whole text has been stored to a file. Typically, `{`⟨*arg2*⟩`}` involves reading the file one or more times, and acting on it in any way we like. The optional argument is used for hacks such as changing where `cprotect` writes his files.

3 Known bugs/limitations

Incompatibility with `\pagestyle{headings}`: when a chapter title is put as a header, it gets upper-cased. If you did `\cprotect\chapter{...}` as usual, the title has been stored to a file, but now, the name of the file is capitalized, and TEX cannot find it.

Issues with nesting of `\cprotect` in `align` environments: the issue seems to arise because the `align` environment wants to fully expand it argument once before typesetting it.

For commands with two or more arguments, it is only possible to put verbatim in one of the arguments (and the syntax is not great).

The argument of any command that is prefixed with `\cprotect` has to have balanced braces, *even when hidden inside verbatim environments, or even comments*. For instance,

```
\cprotect\footnote{On the \verb:{: character}
```

would fail: `\cprotect` would see the brace in `\verb:{:`, and count it as an opening tag, which then has to be closed. This is most likely to lead `\cprotect` to gobble the whole file before complaining. Similarly,

```
\cprotect\footnote{On the \verb:!: %should it be }?
character}
```

would only gobble until the closing brace in `%should it be }?`, and I am not sure what error would be produced.

But we can use one ailment to cure the other! A correct way to typeset the first example is

```
\cprotect\footnote{On the \verb:{: character%}
}
```

if % is a comment character at the time it is read. A safer solution would be to use `\iffalse}\fi` instead of `%}`. It still requires to be sure that `\` is an escape character when this piece of code is read, and can lead to problems if the previous token is `\let` for instance.

4 The code

⟨*package⟩

4.1 Setting up

We first load a few packages

```
\RequirePackage{ifthen}
\RequirePackage{suffix}
```

Then we introduce the commands pertaining to writing files.[4]

We write files `\jobname-1.cpt`, `\jobname-2.cpt`, etc. in order.

```
\newwrite\CPT@WriteOut
\newcounter{CPT@WriteCount}
\newcommand{\CPT@Write}[1]{%
  \stepcounter{CPT@WriteCount}%
  \immediate\openout\CPT@WriteOut=\jobname-%
                                \arabic{CPT@WriteCount}.cpt%
  \newlinechar`\^^M%
  \immediate\write\CPT@WriteOut{#1}%
  \immediate\closeout\CPT@WriteOut%
  \aftergroup\CPT@setLastFileName%
}
\newcommand{\CPT@setLastFileName}{%
  \def\CPT@lastFileName{\jobname-\arabic{CPT@WriteCount}.cpt}}
\newcommand{\CPT@input@last}{%
  \expandafter\protect\expandafter\input
                                \expandafter{\CPT@lastFileName}}
```

The next command changes all catcodes to letters. It was adapted from `filecontents.sty`.

```
\newcommand{\makeallletters}{%
  \count0=0\relax %
  \loop %
```

[4] To be rewritten. For the moment, we use a new file each time cprotect is used. Thus, many files. But this is needed if people want to nest cprotect's.

```
  \catcode\count0=11\relax %
  \advance\count0 by 1\relax %
  \ifnum\count0<256 %
  \repeat %
}
```

4.2 \ReadVerbatimUntil

Both `\ReadVerbatimUntil` and its starred version (which we define using the `suffix.sty` package) take one optional argument `[`⟨*first-cs*⟩`]` and a mandatory argument `{`⟨*final-cs*⟩`}`. The ⟨*final-cs*⟩ is saved as `\CPT@commandatend` to be executed when we close the file (and the group).[5]

```
\newcommand\ReadVerbatimUntil[2][]{%
  \def\CPT@commandatend{#2}%
  \begingroup #1%
  \makeallletters%
  \CPT@setup}
\WithSuffix\newcommand\ReadVerbatimUntil*[2][]{%
  \def\CPT@commandatend{#2}%
  \begingroup #1%
  \makeallletters%
  \CPT@starsetup}
```

`\CPT@setup` reads the four "verbatim" arguments following `{`⟨*final-cs*⟩`}`, and stores them in this order as macros `\CPT@preText`, `\CPT@postText`, `\CPT@begin`, and `\CPT@end`. The macros which read each of these arguments need to be defined inside `\CPT@setup`, because I don't want any constraint on the delimiter. I could write a single macro that gobbles all four arguments at once, but this would require a crazy number of `\expandafter`s, so instead I do it one by one.

If the delimiter was given, say `^`, then we would define `\CPT@readBegin` as `\def \CPT@readBegin#1^{\def \CPT@begin{#1}\CPT@readEnd}`. But since `^` is not given explicitly, we need `\expandafter`s to expand it before the definition takes place. To avoid code repetition, I did it once and for all in the auxiliary macro `\CPT@def`. Note that a parameter of `##1` is somehow hidden inside `\CPT@def`, and that the `##1` inside the replacement text refer to the arguments of the `\CPT@read...` macros.

```
\newcommand{\CPT@def}[2]{\expandafter\def\expandafter#1%
```

[5] It is not a straightforward `\aftergroup`, because I want this to be executed after another `\aftergroup` that comes later.

```
  \expandafter##\expandafter1#2}
\newcommand{\CPT@setup}[1]{%
  \def\CPT@delimiter{#1}%
  \CPT@def\CPT@readPreText\CPT@delimiter{%
       \def\CPT@preText{##1}\CPT@readPostText}%
  \CPT@def\CPT@readPostText\CPT@delimiter{%
       \def\CPT@postText{##1}\CPT@readBegin}%
  \CPT@def\CPT@readBegin\CPT@delimiter{%
       \def\CPT@begin{##1}\CPT@readEnd}%
  \CPT@def\CPT@readEnd\CPT@delimiter{%
       \def\CPT@end{##1}\CPT@readContent}%
  \CPT@readPreText%
}
\newcommand{\CPT@starsetup}[1]{\CPT@setup#1#1#1}
```

We also give the variant `\CPT@starsetup`, which has empty `\CPT@preText` and `\CPT@postText`.

When `\CPT@setup` is expanded, it will call `\CPT@readPreText`, `\CPT@readPostText`, `\CPT@readBegin`, and `\CPT@readEnd`, and finish with `\CPT@readContent`, which we describe now.

We borrow the idea of quark from `expl3`: `\CPT@qend` expands to itself, useful for `\ifx` comparisons. The counter `CPT@numB` will count the surplus of `begin-tag`s compared to `end-tag`s when we parse the text following `\CPT@readContent`. And `\CPT@store` is a macro that adds its argument to an other macro (I was too lazy to learn about token lists). The storage itself will be initialized later.

```
\def\CPT@qend{\CPT@qend}
\newcounter{CPT@numB}
\newcommand{\CPT@store}[1]{\edef\CPT@storage{\CPT@storage#1}}
```

The macro `\CPT@readContent` is quite tricky: if the `begin-tag` and `end-tag` were one character, things would be easy: I would read one character at a time, and compare it to both `begin-tag` and `end-tag`, then either store it, and possibly increase or decrease `CPT@numB`, or decide that I am done if `CPT@numB` becomes negative.

Unfortunately, I want to use `\ReadVerbatimUntil` for environments, in which case the `begin-tag` is `\begin{myenv}` and the `end-tag` is `\end{myenv}`. So two options:

- code a standard string searching algorithm... I did not feel like it, but it might lead to a regexp package later on;
- use TEX's delimited parameters.

I did the latter, using `\CPT@def` again (we want to expand the string which delimits the parameter before doing the definition).

The details are ugly:

- gobble until the first `end-tag`,[6] and insert a fake `begin-tag`, as well as the quark guard `\CPT@qend`,
- inside what we gobbled, gobble `begin-tag`s until reaching the fake one (marked by the quark guard).
- continue until we have one more `end-tag` than `begin-tag`.

```
\newcommand{\CPT@readContent}{%
  \CPT@def\CPT@gobbleOneB\CPT@begin##2{%
    \ifx\CPT@qend##2\CPT@store{##1}\addtocounter{CPT@numB}{-1}%
    \else\CPT@store{##1\CPT@begin}\stepcounter{CPT@numB}%
    \expandafter\CPT@gobbleOneB\expandafter##2\fi}%
  %
  \CPT@def\CPT@gobbleUntilE\CPT@end{%
    \edef\CPT@tempi{##1\CPT@begin}%
    \expandafter\CPT@gobbleOneB\CPT@tempi\CPT@qend%
    \ifthenelse{\value{CPT@numB}<0}{%
      \CPT@store{\CPT@postText}%
      \CPT@Write{\CPT@storage}\endgroup%
      \CPT@commandatend%
    }{%
      \CPT@store{\CPT@end}\CPT@gobbleUntilE%
    }%
  }%
  \setcounter{CPT@numB}{0}%
  \def\CPT@storage{\CPT@preText}%
  \CPT@gobbleUntilE%
}
```

4.3 For macros: `\cprotect` and friends

Equipped with `\ReadVerbatimUntil`, we are ready for the more practical macros. `\cprotect` cheats: it uses `{` and `}` as a `begin-tag` and `end-tag`.

[6] This will fail for devious cases: if `begin-tag` is `abc` and `end-tag` is `bcd`, and `\CPT@readContent` is followed by `abcd...`: we will wrongly see `bcd` as an `end-tag`.

This works most of the time, but fails in cases such as those presented in Section 3 (in usual cases there are workarounds[7]).

The first argument of `\cprotect` is the control sequence `\cs` that we are patching.

```
\newcommand{\cprotect}[1]{\def\CPT@cs{#1}%
  \afterassignment\CPT@A\let\CPT@next}
```

Then, we check whether the next token is a brace or not:

- If it is, we discard the `{`, and the argument of `\cs` stops at the matching explicit closing brace `}`. (See `\CPT@n{` below... yes, the name of the control sequence does contain the brace, which is simply a letter, after all.)
- If it is not, the argument of `\cs` is this token only: we insert a brace after the token, and launch `\ReadVerbatimUntil`. Think of this as inserting braces around the argument, and doing what we did in the former case: discarding the opening brace. (See `\CPT@N` below.)

Since `\ReadVerbatimUntil` makes everything into letters, we need braces to be letters when we define most macros in this Section. We thus need a few `\catcode` changes. Think of `{` → `(` and `}` → `)`.

```
\begingroup
  \catcode`\{=11 \catcode`\}=11 \catcode`\+=11
  \catcode`\(=1 \catcode`\)=2
  \gdef\CPT@A{%
    \ifx\CPT@next\bgroup%
    \expandafter\CPT@n{%
    \else%
    \expandafter\CPT@N%
    \fi)
  \gdef\CPT@n{(\ReadVerbatimUntil*(%
        \CPT@cs(\CPT@input@last))+{+}+)%
  \gdef\CPT@N(\expandafter\CPT@n{\CPT@next })%
\endgroup
```

Note the use of `\bgroup` in `\CPT@A`: if the argument of `\cs` starts with an opening brace, it has been read early, and its `\catcode` will still be `1`.[8]

[7] But I could definitely not have the contents of this `.dtx` file as a (huge) footnote in some document: since `{` and `}` change `\catcode`s, it is unlikely that the numbers balance correctly.

[8] I am not sure whether catcodes really matter in such an `\ifx`.

We use + as a delimiter for \ReadVerbatimUntil. No begin-text nor end-text. begin-tag and end-tag are { and } as mentionned before.

Once the matching } is found, we apply \CPT@cs (\cs stored) to \CPT@input@last, aka the file where we saved the argument (we add some protection so that things work well in \section and similar situations).

Finally, the \cMakeRobust command is a mess, and could probably be improved, although... it works :).

```
\newcommand{\cMakeRobust}[1]{%
  \def\CPT@cs@name{\expandafter\@gobble\string#1}%
  \expandafter\let\csname CPT@old@\CPT@cs@name\endcsname #1%
  \expandafter\def\csname\CPT@cs@name\endcsname{%
    \expandafter\cprotect\csname CPT@old@\CPT@cs@name\endcsname}%
}
```

4.4 For Environments: \cprotEnv\begin and \CPTbegin

\CPTbegin

We introduce the command \CPTbegin, which has a behaviour close to the behaviour of \begin. Namely, \CPTbegin{env} gobbles its argument until it sees the matching \end, and it writes what it gobbled to a file. It then inputs the file between \begin{...} and \end{...}, as we can see from the definition of \CPT@commandatend.

As for the case of \cprotect, we need to setup the values of the four arguments of \ReadVerbatimUntil before reading the content. Since the begin-tag and end-tag depend on a parameter, it would be incredibly messy to try to expandafter the right things, so we define \CPT@env@setup to, well, setup the values of the four arguments of \ReadVerbatimUntil, and then skip directly to \CPT@readContent.

```
\newcommand{\CPTbegin}[1]{%
  \def\CPT@commandatend{\begin{#1}\CPT@input@last\end{#1}}%
  \begingroup%
  \CPT@env@setup{#1}%
  \makeallletters%
  \CPT@readContent%
}
```

As announced, \CPT@env@setup, defined with lots of catcode changes. Since the catcode of \ changes, I need an extra escape character, which I take to be /. I use two groups so each group is opened and closed using the same escape character (this is technically irrelevant, but seems less messy).

```
\begingroup\catcode`\/=0
```

```
/begingroup/catcode`/\=11
  /catcode`/{=11 /catcode`/}=11 /catcode`/-=11
  /catcode`/(=1 /catcode`/)=2
  /gdef/CPT@env@setup#1(%
    /def/CPT@preText(\relax )%
    /def/CPT@postText(\relax )%
    /def/CPT@begin(\begin{#1})%
    /def/CPT@end(\end{#1})%
    )
/endgroup
\endgroup
```

A final piece of code is needed so that `\CPT`-environments can be nested: in order to be able to nest, we need `\ReadVerbatimUntil` to be aware that our environment has been opened when `\CPTbegin{foo}` or `\begin{foo}` appear in the text. Given the rules for delimited parameters in TeX, this is quite difficult. A workaround is to replace `\CPTbegin` by something that ends in `\begin` shuch as `\cprotEnv\begin` as defined below. Thus, for nesting to work, you need to prepend every one of your `env` environments with `\cprotEnv`.

```
\def\cprotEnv\begin{\CPTbegin}
```

⟨/package⟩

Change History

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.