The codeverbatim Package
Version 1.0

Alceu Frigeri*

November 2026

Abstract

This package enables the construction of custom verbatim commands/environments, allowing the customization of the desired behaviour, including (pre/post) processing of the grabbed code before typesetting it.

Contents

1 Introduction **1**

2 The codeverbatim Package **2**
2.1 Characters Category Code and Common Keys . . . 2
2.2 New Verbatim Environment . . . 3
2.2.1 catcode commands . . . 3
2.3 Store commands . . . 4
2.4 Retokenizing . . . 5
2.5 Display commands . . . 5
2.6 Short Verbatim . . . 6
2.6.1 Caveats . . . 7

3 Examples **7**
3.1 Preloading sets . . . 7
3.2 A minimal construct . . . 8
3.3 Customizing the language . . . 8
3.4 Further customization . . . 8
3.5 Using sequences . . . 9
3.6 Verbatim shortcuts . . . 9
3.6.1 Custom settings . . . 10
3.6.2 Custom highlight set . . . 10
3.6.3 Custom shortcuts . . . 10
3.7 A floating environment . . . 11
3.8 Using listings with codeverbatim . . . 12

4 To do, Future plans **13**

1 Introduction

The aim of this package (and its companion *codehighlight* [2]) is to allow the creation of fully customizable verbatim environments. To achieve that, unlike other packages of the kind as *fancyvrb* [9], *scontents* [4] or *listings* [5], which have a fixed set of ways to collect/display a verbatim environment, *codeverbatim* and *codehighlight* split the action of grabbing some text in verbatim mode (this package, *codeverbatim*) from typesetting it with or without syntax highlight (package *codehighlight* [2]).

codeverbatim package offers a mechanism to create custom verbatim environments whereas the grabbed text will be stored either in a *expl3* [8] token list or sequence variables with hooks allowing the customization of the entire process.

*https://github.com/alceu-frigeri/codeverbatim

Note: A great deal of the functionality of this package is inspired on the *scontents* [4] package.

Note: Although the two packages are though to be used together, they can be used separately. It's perfectly possible to use *codeverbatim* together with *listings* [5] package, or *codehighlight* [2] with other verbatim package, like *fancyvrb* [9] (using the macros `\SaveVerb` and `\UseVerb`) or *scontents* [4] (redefining the *verbatimsc* environment).

Lastly, this package is fully written in *expl3* [8], but given that one might want to use it in a LaTeX 2_ε context (either at package or document level), all public commands have a LaTeX 2_ε alias, and, for simplicity, they are listed together.

2 The codeverbatim Package

The aim of this package is to allow to construction of custom environments and commands to capture verbatim content. Among others, one can customize:

- which characters should be captured with category code other.
- how to handle the text after the environment begin and the first end of line.
- how to process the captured code.

Another point is that this package only offers ways to capture content in verbatim mode, leaving it to the user to define how to pre/postprocess and display it, for instance, using *codehighlight* [2] package (see section *Display commands*).

2.1 Characters Category Code and Common Keys

When capturing content in verbatim mode, many characters have their usual category code changed to other (12), but not all. In particular `^^M` (char: 13, End Of Line character) always has its category code changed to other.

`\`	(char: 92)	`$`	(char: 36)	`^^A`	(char: 1)
`{`	(char: 123)	`^`	(char: 94)	`^^I`	(char: 9)
`}`	(char: 125)	`_`	(char: 95)	`^^J`	(char: 10)
`#`	(char: 35)	`&`	(char: 38)	`^^K`	(char: 11)
`%`	(char: 37)	`~`	(char: 126)		(char: 32, space)

Note: This list is named "other characters list" in the following (for simplicity).

Note: As noted above, `^^M` will always have its category code changed to other (12). Moreover, following the `\tl_retokenize:n` convention, lines will be terminated by `^^J` (and not `^^M`), also with category code other. See the chapter "Reassigning category codes" (under "Manipulating tokens in token lists") from *The LaTeX3 Interfaces* [8], for details.

By default no other character has its category code changed. This can be changed with the keys below, in which there are two cases:

(i) List of characters set to other: keys *set other* and *other*,

(ii) List of characters set to letter: key *letter*,

These keys are common to all main code verbatim commands and expect a comma separated list of characters:

`set other` This will reset and fully replace the "other characters list". If used more than once, only the last attribution remains.

The default is the "other characters list" above. Don't forget to escape the characters as needed, for instance, like `\^^K`

`other` (default: empty list) This will add characters to the "other characters list" and can be called multiple times.

`letter` (default: empty list) The characters on the list (cumulative list) will be made letter.

Note: These character lists shall contain a single char per item! and they will raise an error if more than one character (per item) is found.

Attention If all keys are given, first *other* then *letter* will be applied.

2.2 New Verbatim Environment

\cvNewEnvironment `\cvNewEnvironment [⟨keys⟩] {⟨env-name⟩} {⟨arg spec⟩} {⟨begin-code⟩} {⟨end-code⟩}`
\codeverb_new_environment:nnnnn `\codeverb_new_environment:nnnnn {⟨keys⟩} {⟨env-name⟩} {⟨signature⟩} {⟨begin-code⟩} {⟨end-code⟩}`

This will create a new document environment ⟨env-name⟩, following the `\NewDocumentEnvironment` structure. ⟨arg spec⟩ is the desired ⟨env-name⟩ parameter specification list (`\NewDocumentEnvironment` 1st parameter, see *LaTeX for authors* [7]). The ⟨begin-code⟩ code will be executed before the verbatim code gets grabbed. The ⟨end-code⟩ will be executed right after, see the keys *inline* and *verb tl* below for details on how to further handle the verbatim content.

> ***Note:*** If short verbatim commands are on (`\cvSVerbOn` or `\codeverb_sverb_on:`) those active characters will be made other (see 2.6.1) after applying the characters category code defined here (with the common keys from 2.1) and before applying the characters category code defined with `\cvSetEnvCC` or `\codeverb_set_env_catcode:n` (see 2.2.1 below).

⟨keys⟩ can be any combination of (besides the ones from 2.1):

`drop first` (default: true) If the character following the environment begin until the line end shall be processed or discarded. The default behaviour is similar to other packages of the kind, like *listings* or *fancyvrb*

`drop blanks` (default: false) If set to true, and the initial lines are blank, those lines will be discarded.

`line proc` (default: -nothing-) The given command shall grab 2 parameters. It will receive a single line (verbatim material) as the 2nd parameter and a temporary token list as the 1st parameter. The simplest command would be `\tl_set:Nn` (no pre-processing at all). This can be used to pre-process the verbatim content, one line at a time.

`inline` (default: false) If set, ⟨end-code⟩ will receive the verbatim material as `#1`. This has precedence over the key *verb tl*.

`verb tl` (default: `\g_codeverb_verb_body_tl`) If *inline* is set to false, the verbatim material will be stored in the given token list variable, before ⟨end-code⟩ is executed. The assignment is global.

Off course, the default is ok to be used in an *expl3* [8] code régime, you may want to change that.

2.2.1 catcode commands

\cvSetEnvCC `\cvSetEnvCC {⟨keys⟩}`
\codeverb_set_env_catcode:n `\codeverb_set_env_catcode:n {⟨keys⟩}`

These can only be used in the declaration of a new verbatim environment, and are meant to be used in the ⟨begin-code⟩, allowing for further (local) customization of characters category code. ⟨keys⟩ can be any combination of the common keys (see 2.1). For example:

```
\cvNewEnvironment[drop first]{myEnv}{O{}}
  { \cvEnvCCset {#1} }
  { }
...

\begin{myEnv}[letter = { @ , \_ }]
  some text
\end{myEnv}
```

Note that the environment *myEnv* was defined with the default verbatim setup (category codes as in 2.1). The single optional parameter allows the end user to change the category code of characters.

> ***Note:*** It won't replace the environment defaults, but will be applied after the environment setup.
>
> ***Attention*** Unlike in all other cases, the default "other characters list" for these commands is an empty list. That way, it will only change the catcode (to other or letter) of the given characters.

2.3 Store commands

Besides just storing the verbatim content in a token list variable (key *verb tl* above) one has a few other options:

(i) store the verbatim content in dedicated sequence variables (similar to *scontents* [4] package),
(ii) display them using another package (like *listings* [5]) or
(iii) use them as latex code right away. The following commands aim at exactly that.

\cvNewStoreCommands	\cvNewStoreCommands [⟨key⟩] {⟨cs-base⟩}
\codeverb_new_store_cmds:nn	\codeverb_new_store_cmds:nn {⟨key⟩} {⟨cs-base⟩}

These will create a set of commands prefixed with ⟨cs-base⟩, using a set of dedicated *expl3* [8] sequence variables to help handle verbatim code in a way similar to how the package *scontents* [4] does. By default, two *expl3* [8] commands and two LaTeX 2ε aliases will be created. This can be changed with the following key:

`expl3` (default: false) If set, only the *expl3* [8] variants will be created and no LaTeX 2ε aliases will be defined.

\<cs-base>PutRight	\<cs-base>PutRight {⟨st-name⟩} {⟨verb-content⟩}
\<cs-base>PutRightnV	\<cs-base>PutRight {⟨st-name⟩} {⟨tl-var⟩}
\<cs-base>Merge	\<cs-base>Merge {⟨var⟩} {⟨base-list⟩} {⟨index-list⟩}
\<cs-base>_gput_right:nn	\<cs-base>_gput_right:nn {⟨st-name⟩} {⟨verb-content⟩}
\<cs-base>_gput_right:nV	\<cs-base>_gput_right:nn {⟨st-name⟩} {⟨tl-var⟩}
\<cs-base>_merge:Nnn	\<cs-base>_merge:Nnn {⟨var⟩} {⟨base-list⟩} {⟨index-list⟩}

The \<cs-base>PutRight (or \<cs-base>_gput_right:nn) will create (as needed) an internal sequence variable associated with ⟨st-name⟩ and store ⟨verb-content⟩ in it. The \<cs-base>PutRightnV and \<cs-base>_gput_right:nV variants will store the content of a token list var, ⟨tl-var⟩.
The \<cs-base>Merge (or \<cs-base>_merge:Nnn) will merge items of one, or more, of those sequences.

The syntax choosen resembles the one from the command \mergesc from package *scontents* [4], and there are two main (proper) cases:

- ⟨index-list⟩ is an empty list.

  In this case ⟨base-list⟩ is expected to be a list of the kind ⟨st-name⟩ [⟨index-ref⟩], where ⟨index-ref⟩ can be a single number (the sequence index) or a range like ⟨index⟩-⟨index⟩. 1 is the index of the first item, while *end* can be used to reference the last item of a sequence, so that, ⟨st-name⟩[2-end] will refer all items starting from the 2nd. If no [⟨index-ref⟩] is given, then the last item of ⟨st-name⟩ will be used.

- ⟨index-list⟩ isn't empty.

  In this case ⟨base-list⟩ is expected to be a list of ⟨st-name⟩s (no indexes given), while ⟨index-list⟩ is expected to be a list of just indexes, then all indexes from ⟨index-list⟩ will be applied to each ⟨st-name⟩ from ⟨base-list⟩.

For example:

```
\cvNewStoreCommands {myStore}
...
... many calls to \myStorePutRight { case-A } { -content- }
...
... many calls to \myStorePutRight { case-B } { -content- }

\myStoreMerge { \tempA } { case-A[1] , case-B[2] , case-A[2-end] , case-B }

\myStoreMerge { \tempB } { case-A , case-B } { 1 , 3-end }
```

\tempA will end up with the 1st item from ⟨case-A⟩, then the 2nd item from ⟨case-B⟩, followed by all items from the 2nd to the last one from ⟨case-A⟩, and finally the last one from ⟨case-B⟩.

\tempB will end up with the 1st item from ⟨case-A⟩ followed by all items from the 3rd to the last one from ⟨case-A⟩, then the 1st item from ⟨case-B⟩ followed by all items from the 3rd to the last one from ⟨case-B⟩.

2.4 Retokenizing

`\cvRetokenEnvNewCommand`	`\cvRetokenEnvNewCommand` [⟨key⟩] {⟨env-cmd⟩} {⟨env-name⟩}
`\codeverb_retoken_env_newcs:nnn`	`\codeverb_retoken_env_newcs:nnn` {⟨key⟩} {⟨env-cmd⟩} {⟨env-name⟩}

These will create a set of commands (see below) tailored at retokenizing some (verbatim) content inside the given environment ⟨env-name⟩. By default, two *expl3* [8] commands and two LaTeX 2_ε aliases will br created. This can be changed with the following key:

`expl3` (default: false) If set, only the *expl3* variants will be created and no LaTeX 2_ε aliases will be defined.

`\<env-cmd>`	`\<env-cmd>` {⟨verb-content⟩}
`\<env-cmd>V`	`\<env-cmd>V` {⟨tl-var⟩}
`\<env-cmd>:n`	`\<env-cmd>:n` {⟨verb-content⟩}
`\<env-cmd>:V`	`\<env-cmd>:V` {⟨tl-var⟩}

`\<env-cmd>` (viz-à-viz `\<env-cmd>:n`) will re-scan (retokenize) ⟨verb-content⟩ inside an ⟨env-name⟩ environment. Similarly, `\<env-cmd>V` (viz-à-viz `\<env-cmd>:V`) will re-scan the contents (single expansion) of the ⟨tl-var⟩ inside the ⟨env-name⟩ environment. So that, for example:

```
\lstnewenvironment{lstenv}
  {
    % listings setup
  }
  {}

\cvRetokenEnvNewCommand
  {VerbListing} {lstenv}
```

once defined, `\VerbListingV` `\g_codeverb_verb_body_tl` (err, the variable defined with the *verb tl*) could be called in the ⟨end-code⟩ of a verbatim environment (see 2.2), which would result in the verbatim content just grabbed (into `\g_codeverb_verb_body_tl`) being displayed using the *lstenv* environment (set up with the *listings* [5] package). This will be, effectively, equivalent to

```
\begin{lstenv}
  ... content of \g_codeverb_verb_body_tl 'retokenized' ...
\end{lstenv}
```

Important: The verbatim code will be re-scanned (retokenized) in the process with the current category code régime.

`\cvRetokenNewCommand`	`\cvRetokenNewCommand` {⟨cmd⟩}
`\cvRetokenNewCommandV`	`\cvRetokenNewCommandV` {⟨cmd⟩}
`\codeverb_retoken_newcs:N`	`\codeverb_retoken_newcs:N` {⟨cmd⟩}
`\codeverb_retoken_newcsV:N`	`\codeverb_retoken_newcsV:N` {⟨cmd⟩}

`\cvRetokenNewCommand` and `\codeverb_retoken_newcs:N` will make ⟨cmd⟩ an alias to `\tl_retokenize:n` (from *expl3* [8]), while `\cvRetokenNewCommandV` and `\codeverb_retoken_newcsV:N` will make ⟨cmd⟩ an alias to `\tl_retokenize:V`. These commands are just for completeness (with the previous set of commands, which retokenize some verbatim content inside a given environment). Moreover, their main aim is to allow the code, grabbed as verbatim content, to be (re)used as LaTeX commands/text, the same overall idea from the commands `\getstored` (package *scontents* [4]) or `\tsexec` (package *codedescribe* [1]).

2.5 Display commands

`\cvHideSpc`	`\cvHideSpc` {⟨text⟩}
`\codeverb_hide_spc:n`	`\codeverb_hide_spc:n` {⟨text⟩}

These will just replace every occurrence of a space (character code 32) by `\space`. They are the default commands used by `\cvSVerb` and `\codeverb_sverb:Nw` and are needed because (by default) spaces also have their category code changed to other (see 2.1), resulting that spaces are visible as in "example␣of␣visible␣spaces".

2.6 Short Verbatim

The following commands aims at defining "short cuts" (active characters, as in other verbatim packages) allowing to grab short verbatim content and use it "right away", for instance, inside a paragraph. Unlike other verbatim packages, the following commands allow to customize the used characters category code, per command, as well to customize the action taken with the just grabbed verbatim content.

`\cvSVerbSet`	`\cvSVerbSet` [⟨keys⟩] {⟨code⟩}
`\cvSVerb`	`\cvSVerb` {⟨char⟩} ... ⟨verbatim content⟩...⟨char⟩
`\codeverb_sverb_setcs:nn`	`\codeverb_sverb_setcs:nn` {⟨keys⟩} {⟨code⟩}
`\codeverb_sverb:Nw`	`\codeverb_sverb:Nw` {⟨char⟩} ... ⟨verbatim content⟩...⟨char⟩

Both, `\cvSVerb` and `\codeverb_sverb:Nw`, are the immediate commands. They will use the character, ⟨char⟩, following them, as a marker and grab everything between it and its next appearance as verbatim content. ⟨code⟩ (set with `\csSVerbSet` or `\codeverb_sverb_setcs:nn`) will receive the verbatim content as `#1`. The default ⟨code⟩ is `\codeverb_hide_spc:n {#1}` "hidden" spaces.

> ***Note:*** The category code of ⟨char⟩ itself will be made other, regardless of the actual underlying category code in use whilst grabbing the content, the category code change being local.

With `\csSVerbSet` and `\codeverb_sverb_setcs:nn` one can customize the commands `\cvSVerb` and `\codeverb_sverb:Nw`. ⟨keys⟩ can be any of the common keys (see 2.1) and the following:

`no endmark` This will suppress the ending mark, `^^J`, which is introduced by default (following the `\tl_retokenize:n` convention, see *"Reassigning category code"* from *expl3* [8])

⟨code⟩ will receive the grabbed content as `#1`. For instance, with `\cvSVerbSet[no endmark]{#1}` the content grabbed with `\csSVerb` will be presented "as is" with visible spaces. Likewise, `\cvSVerbSet{ { \ttfamily #1 } }` will result that the content grabbed with `\csSVerb` will be presented with the `\ttfamily` (note the extra set of braces to localize the effect of the `\ttfamily` command).

> ***Note:*** One can use `\hlDisplay` or `\codehl_display:nn` (see the section *Display commands* from *codehighlight* [2]) when defining ⟨code⟩, for instance: `\cvSVerbSet { \hlDisplay[inline]{#1} }`.
>
> ***Warning:*** Those display commands (from *codehighlight* [2]) expect the content to be terminated with a `^^J`. Do not use the *no endmark* with them, otherwise a quark loop will ensue.

`\cvSVerbNew`	`\cvSVerbNew` [⟨keys⟩] {⟨start-patt⟩} {⟨end-patt⟩} {⟨code⟩}
`\cvSVerbOn`	`\cvSVerbOn`
`\cvSVerbOff`	`\cvSVerbOff`
`\codeverb_sverb_new:nnnn`	`\codeverb_sverb_new:nnnn` {⟨keys⟩} {⟨start-patt⟩} {⟨end-patt⟩} {⟨code⟩}
`\codeverb_sverb_on:`	`\codeverb_sverb_on:`
`\codeverb_sverb_off:`	`\codeverb_sverb_off:`

`\cvSVerbNew` and `\codeverb_sverb_new:nnnn` allow the setup and definition of custom short commands associated with active chars (characters with category code active, 13). Once defined, and activated in text, everything between ⟨start-patt⟩ and ⟨end-patt⟩ will be grabbed as verbatim content and processed by ⟨code⟩ (which will receive the verbatim content as `#1`. These can be called more than once, defining different ⟨star-patt⟩, ⟨end-patt⟩, and associated ⟨code⟩. `\cvSVerbOn` and `\codeverb_sverb_on:` turn those definitions on, while `\cvSVerbOff` and `\codeverb_sverb_off:` deactivate them. For example:

```
\ExplSyntaxOn
\codeverb_sverb_new:nnnn { no ~ endmark } {|*}
      {*|}
  { \tl_trim_left_spaces:n { #1 } }
\ExplSyntaxOff

\cvSVerbOn
Example of use: |* \cmd{#1}*|
\cvSVerbOff
```

Example of use: \cmd{#1}

> ***Note:*** ⟨keys⟩ are the same from the `cvSVerbSet` and `\codeverb_sverb_setcs:nn`. The same remarks and warning (regarding the *no endmark* key) applies.

2.6.1 Caveats

A few caveats to be considered:

- ⟨start-patt⟩ is a single (unique) character. This is the simplest case, the ⟨start-patt⟩ character will be made active and the content between it and ⟨end-patt⟩ will be collected as verbatim content.
- ⟨start-patt⟩ is a pattern compose of more than one character. In this case the 1st character of ⟨start-patt⟩ will be made active and look ahead to see if the following tokens match the rest of the ⟨start-patt⟩, if so, then, the content between ⟨start-patt⟩ and ⟨end-patt⟩ will be collected as verbatim content.
- There are more than one ⟨start-patt⟩ with the same 1st character: This is very similar to the previous case, though, the active code (associated with the 1st character) will test for every possible ⟨start-patt⟩, starting by the last defined one.

  Note: For instance, one may want 3 definitions for ⟨start-patt⟩: `|`, `||` and `|||`, in which case the longest one should be defined last.

  Attention When looking ahead, `\peek_regex:NTF` is used. One side effect is that the first character following the ⟨start-patt⟩ will be grabbed with the current category code table (and not the verbatim one, see *"Peeking ahead at the next token"* from *expl3* [8]). In particular if the 1st character following ⟨start-patt⟩ is a special one, like `\` or `#`, this will, probably, lead to low level errors. The work around is to use a single space after the ⟨start-patt⟩, since this 1st space will be grabbed with the current category code it can be easily removed with a command like `\tl_trim_left_spaces:n` (as in the previous example) since it will (default settings) be the single space with category code 10 (space).
- The characters made active by the `\cvSVerbNew` and `\codeverb_sverb_new:nnnn`, while effectively actives (`\cvSVerbOn`), will be made other when using a verbatim environment (see 2.2) or using any of the short verbatim commands.

`\cvSVerbUndef`	`\cvSVerbUndef` {⟨start-patt⟩}
`\cvSVerbUndefAll`	`\cvSVerbUndefAll`
`\codeverb_sverb_undef:n`	`\codeverb_sverb_undef:n` {⟨start-patt⟩}
`\codeverb_sverb_undef_all:`	`\codeverb_sverb_undef_all:`

`\cvSVerbUndef` and `\codeverb_sverb_undef:n` will undefine the short cut associated with ⟨start-patt⟩, while `\cvSVerbUndefAll` and `\codeverb_sverb_undef_all:` will undefine all current defined short cuts.

3 Examples

In the following examples, for simplicity, mostly LaTeX 2_ε commands are used. Furthermore, they are based on the commands from this package and the ones from *codehighlight* [2], since this package alone has no provision for syntax highlight.

Moreover, for simplicity (and keeping them in sync) this section is duplicated in both packages, resulting in both packages being cited, `\cite{}`, instead of using internal links, `\ref{}`.

3.1 Preloading sets

In the preamble (see *"Pre-defined sets"* from *codehighlight* [2]):

```
% only in the preamble
\hlLangLoad{ C , php , TeX }
```

This will trigger the loading of the files associated with *C*, *php* and *TeX*. Note that, since *php* is based on *html* the file associated with it will also be loaded.

Warning: Keep in mind that loading of files can only happen in the preamble, otherwise an error will ensue.

Note: This isn't mandatory if, in the preamble, one uses `\hlSettings`, referencing the desired language/set or `\hlLangDef` / `\hlKeysetDef`, either extending or based on the desired language/set. Those commands also trigger the loading of the needed auxiliary files.

3.2 A minimal construct

A minimal construct could entice setting only keywords (but no language, therefore, no comment or string will be recognized), and define an environment which will display the collected verbatim code right away (see *"Display Settings"* from *codehighlight* [2] and *"New Verbatim Environment"* from *codeverbatim* [3]):

```
% if the file definition were already loaded, the following can be anywhere
\hlSettings
  {
    keywd set = { C }
  }

\cvNewEnvironment[verb tl=\DemoV]{DemoEnv}{}
  {}
  {
  \hlDisplayV \DemoV
%  \myShow \DemoV
  }
```

```
  \begin{DemoEnv}
    for ( i=0 , i<10 , i++ ) {
      g += i;
    }
    // a comment
  \end{DemoEnv}
```

```
for ( i=0 , i<10 , i++ ) {
  g += i;
}
// a comment
```

Changing the settings to include the language syntax:

```
\hlSettings
  {
    language = C ,
    keywd set = C
  }
```

```
  \begin{DemoEnv}
    for ( i=0 , i<10 , i++ ) {
      g += i;
    }
    // a comment
  \end{DemoEnv}
```

```
for ( i=0 , i<10 , i++ ) {
  g += i;
}
// a comment
```

3.3 Customizing the language

One can change an existing language definition, for instance, adding a LaTeX escape (the *nestable* key is to allow it to be detected even inside another comment, see *"Language Syntax"* from *codehighlight* [2]):

```
\hlLangDef{C}
  {
    active comment line = {1}{\$\$}{ nestable }
  }
```

```
  \begin{DemoEnv}
    for ( i=0 , i<10 , i++ ) {
      g += i;
    }
    // a comment $$ \textbf{this:} $A_i$
  \end{DemoEnv}
```

```
for ( i=0 , i<10 , i++ ) {
  g += i;
}
// a comment this: A_i
```

3.4 Further customization

One can use color to call attention to some parts of the code, in general this is achieved by customizing a ⟨hlset⟩ (see *"Highlight Sets"* from *codehighlight* [2]). Note that, in the following example, ⟨std-hlset⟩ is a pre-defined ⟨hlset⟩ (starting one).

```
\hlLangDef{C}
  {
    comment line = {2}{//+}{ }
  }
\hlHLsetDef{std-hlset}
  {
    comment = {2}{\itshape\color{red}}
  }
```

```
  \begin{DemoEnv}
    for ( i=0 , i<10 , i++ ) {
      g += i;
    }
    // a comment $$ \textbf{this:} $A_i$
    //+ really important comment
  \end{DemoEnv}
```

```
for ( i=0 , i<10 , i++ ) {
  g += i;
}
// a comment this:  A_i
//+ really important comment
```

3.5 Using sequences

Note, the following example could be written in LaTeX 2ε, but, it's more natural to write it in *expl3* [8] (given the use of sequences). First a dedicated store command is created (see *"Store commands"* from *codeverbatim* [3]) then an environment using it (see *"New Verbatim Environment"* from *codeverbatim* [3]).

```
\ExplSyntaxOn

  \codeverb_new_store_cmds:nn { } { DemoStore }
  \cs_generate_variant:Nn \DemoStore_gput_right:nn { V }

  \codeverb_new_environment:nnnnn
    {
      drop ~ first,
      inline ,       % The environment end code will receive the verbatim code as #1
      other =  { \^^K } ,
    }
    {DemoStore} {O{default}}
    { %% Here, #1 is the environment optional parameter
      \tl_set:Nn \DemoStore_tmp_tl {#1}
    }
    { %% Here, #1 is the collected verbatim code
      \DemoStore_gput_right:Vn
          \DemoStore_tmp_tl
          { #1 }
    }
\ExplSyntaxOff
```

The code above defines a new environment, *DemoStore*, which expects a single optional parameter: the 'sequence name' to be used. Once a few verbatim snips are collected, they can be merged (command `\DemoStoreMerge` below, created by the command `\codeverb_new_store_cmds:nn` above) and then displayed as a simple piece.

```
\begin{DemoStore}[sampleA]
      for ( i=0 , i<10 , i++ ) {
      g += i;
    }    // a comment $$ \textbf{this:} $A_i$
\end{DemoStore}

\begin{DemoStore}[sampleA]
      for ( j=0 , j<10 , j++ ) {
      h += i;
    }    // a comment $$ \textbf{this too:} $B_j$
\end{DemoStore}

\DemoStoreMerge {\mymerged} {sampleA} {1-2}
\hlDisplayV \mymerged
```

```
  for ( i=0 , i<10 , i++ ) {
  g += i;
}    // a comment this:  A_i
  for ( j=0 , j<10 , j++ ) {
  h += i;
}    // a comment this too:  B_j
```

3.6 Verbatim shortcuts

The following is to be used with short verbatim commands (see *"Short Verbatim"* from *codeverbatim* [3]).

```
\hlBasefmtDef { ShortVerb }  %%% all non given keys start with their default values
  { inline }
```

3.6.1 Custom settings

A few dedicated settings, per language:

```
\hlSettingsDef { PHP }
  {
    based on language    = { php }  ,
    based on keywd set   = { html , php }  ,
    based on hl set      = { std-hlset }  ,
    based on base format = { ShortVerb }
  }
```

```
\hlSettingsDef { C++ }
  {
    based on language    = { C++ }  ,
    based on keywd set   = { C , C++ }  ,
    based on hl set      = { std-hlset }  ,
    based on base format = { std-basefmt }
  }
```

```
\hlSettingsDef { C }
  {
    based on language    = { C }  ,
    based on keywd set   = { C }  ,
    based on hl set      = { std-hlset }  ,
    based on base format = { std-basefmt }
  }
```

```
\hlSettingsDef { TeX }
  {
    based on language    = { TeX }  ,
    based on keywd set   = { TeX.common }  ,
    based on hl set      = { std-hlset }  ,
    based on base format = { ShortVerb }
  }
```

Note that, all sets have their own “name space”, You can have a settings named “C++”, a language named “C++”, a keyword set named “C++” and so own. None correlated, or attached, to the other.

3.6.2 Custom highlight set

Besides being an example of a custom highlight set, this is just to emphasize the use of the shortcuts commands in the following examples.

```
\hlHLsetDef { std-hlset }
  {
   keywd  = {1}{\bfseries\color{blue}} ,
   php    = {1}{\bfseries\color{red}}  ,
   html   = {1}{\bfseries\color{green}}  ,
   string = {2}{\slshape\color{white!50!red}}  ,
   string = {1}{\slshape\color{white!50!black}}  ,
  }
```

3.6.3 Custom shortcuts

The following will define 4 shortcuts, one “raw” (showing spaces), one for TEX commands, one for C++ and another one for html/php (see *“Short Verbatim”* from `codeverbatim` [3]).

```
\cvSVerbNew [no endmark]{|}{|}
  {#1}

\cvSVerbNew {||}{||}
  {\hlSettingsSelect{TeX}\hlDisplay[inline]{#1}}

\cvSVerbNew {|<}{>|}
  {\hlSettingsSelect{PHP}\hlDisplay{#1}}

\cvSVerbNew {|(}{)|}
  {\hlSettingsSelect{C++}\hlDisplay[inline]{#1}}
```

Note that, in the first case, since the collected code will be used "raw" (no processing), the *no endmark* is used to suppress the insertion of an end mark, `^^J`, which would, otherwise, be rendered as an small speck, like ˙.

> ***Note:*** If the key *inline* is part of the base format, it is possible to omit it when using `\hlDisplay`.

Using an explicit short verbatim command

```
%% this will set it, plain verbatim text, hidding spaces.
\cvSVerbSet[no  endmark]{\cvHideSpc{#1}}

then you can have \cvSVerb!a \short \one ...! in the middle of some text.
```

then you can have a \short \one ... in the middle of some text.

Or a more elaborated example of use, turning on the short verbatim shortcuts, `\cvSVerbOn`

```
\cvSVerbOn
  A more elaborated case,
  showing spaces,  |quick \and \simple| or
  some \LaTeX{} source code like || this is \TeX{} || followed by
  simple php |< <?php echo "some text"; ?> some other echo >|,
  why not C like |( for ( j=0 , j<10, j++) )| or
  some php inside some html |< <a source="<?php echo "some text"; ?>" > and    >|.
  All in the same paragraph.

\cvSVerbOff
```

A more elaborated case, showing spaces, quick␣\and␣\simple or some LaTeX source code like `this is \TeX{}` followed by simple php `<?php echo "some text"; ?> some other echo` , why not C like `for ( j=0 , j<10, j++)` or some php inside some html `<a source="<?php echo "some text"; ?>" > and  ` . All in the same paragraph.

> ***Note:*** When using these shortcuts, take note that `\peek_regex:NTF` is used to recognize the many commands that start with the same (active) character, which has the side effect that the first following character will be captured with the current code régime. See *"Short Verbatim"* and *"caveats"* from *codeverbatim* [3] for details.

3.7 A floating environment

Using the package *newfloat* [6], one can declare a new floating environment (including a "list of ...") as follow:

```
%% This must be declared in the preamble
\DeclareFloatingEnvironment
  [fileext={lov},listname={List of C code},name={C code},placement=htb]
  {floatenv}
```

Then, it's a matter of declaring a dedicated verbatim environment that uses the floating environment declared above:

```
\ExplSyntaxOn
  \codeverb_new_environment:nnnnn
    {
      drop ~ first,
      inline ,       % The environment end code will receive the verbatim code as #1
    }
    {DemoFloat} {m}
    { %% Here, #1 is the environment optional parameter
      \tl_set:Nn \l_caption_tl {#1}
    }
    { %% Here, #1 is the collected verbatim code
      \begin{floatenv}
        \exp_args:No \caption \l_caption_tl
        \hlSettingsSelect{C}
        \hlDisplay{#1}
      \end{floatenv}
    }
\ExplSyntaxOff
```

Lastly, one can just use it as any other floating environment:

```
\begin{DemoFloat}{C sample}
  for ( i=0 , i<10 , i++ ) {
    g += i;
  }    // a detail $$ $B_j$
\end{DemoFloat}
```

C code 1: C sample

```
  for ( i=0 , i<10 , i++ ) {
    g += i;
  }    // a detail B_j
```

3.8 Using listings with codeverbatim

Another possibility, is using *codeverbatim* [3] together with *listings* [5], that for is enough to set a *listings* environment (see it's manual for details), a "retokenizing" command to re-scan the verbatim code with *listings*, `\cvRetokenEnvNewCommand` below, see *"Retokenizing"* from *codeverbatim* [3], and a verbatim environment which uses the "retokenizing" command:

```
\lstnewenvironment{cvlst}
  { \lstset{language=Pascal} }
  {}

\cvRetokenEnvNewCommand
  {cvlisting} {cvlst}

  \cvNewEnvironment
    [
      drop first = false,
      inline ,       % The environment end code will receive the verbatim code as #1
      other =  { \^^K } ,
    ]
    {DemoListing} {}
    { %% Here, #1 is the environment optional parameter
    }
    { %% Here, #1 is the collected verbatim code
      %% note the preceeding ^^J
      \cvlisting {^^J #1}
    }
```

Note: The `^^J` is to "give" listings a first empty line, so it won't complain about it.

Thereafter, this is like any other listing environment, though, one where one could further customize every aspect of it, like preprocessing the code, using sequences to store the code before retokenizing it, etc..

```
\begin{DemoListing}
 for i:=maxint to 0 do
 begin
 { do nothing }
 end;

 Write('Case insensitive ');
 WritE('Pascal keywords.');
\end{DemoListing}
```

```
 for i:=maxint to 0 do
begin
{ do nothing }
end;

Write('Case insensitive ');
WritE('Pascal keywords.');
```

4 To do, Future plans

These are some aspects (and random ideas) that are either planned or that deserve some consideration:

- grabbing code from an external file (codeverbatim): defining a 'text marker' (begging/end ... label).
- optimize the background color code. It works as is, but, it's slow.
- "here code" typical syntax <<Label or <<'Label'. Possible approach: _scanT collects the next label and sets _end_regex.
- handling of tabs (? fixed spacing or column mod num ?).
- 'simple' identifiers :
  - (maybe) <start>... followed by an explicit regex (expl3): not really an identifier (name it 'markers' ??) like <start> => `#` <regex> `\d \d \d | \d \d | \d` (that would make for `#1` or `#20` or `#250` (Delphi)). That could also be used to highlight 'other keywords' (if they fit a regular expression).
- 'delimited' identifiers :
  - <start> ... <end> (some multi-word compiler directives). code problem: a 'keyword' may contain an space then. phrase keywords.
  - (maybe) delimited identifier followed by regex (for instance, *expl3* signatures). issue: needs a different/unique ⟨start-pattern⟩ (no backtrack).
- (maybe) blank lines at the end (codeverbatim?)... possible, but would need some temp (seq? tl? int?) holding 'blank lines' (delayed insertion) ... waiting to see if something else comes along...main problem: user supplied 'line processing'... before or after it?
- indexing might be implement in a similar way as it is in *codedescribe*.
- allow for further customization of line numbering: e.g. one every 5, or starting number, etc.
- (maybe) change the markers from `\q_nil` to something more end user safe, like `\s_stop` or `\scan_stop:` ("drawback" no pre-existent cmd like `\quark_if_nil:`... but could prevent a quark loop in case of misuse.)
- simula/algol comments peculiar syntax.
- listings *procnamekeys* might be implemented. Idea: add another 'list' (like identifiers) whose 'result' would be setting a flag for the following keyword.
- writing code to an external file, albeit fairly simple, isn't planned for the time being.
- improve the examples section.

References

[1] Alceu Frigeri. *The codedescribe and codelisting Packages*. 2026, p. 21. URL: `https://mirrors.ctan.org/macros/latex/contrib/codedescribe/doc/codedescribe.pdf` (visited on 08/07/2026).

[2] Alceu Frigeri. *The codehighlight Package*. 2026, p. 24. URL: `https://mirrors.ctan.org/macros/latex/contrib/codehighlight/doc/codehighlight.pdf` (visited on 11/30/2026).

[3] Alceu Frigeri. *The codeverbatim Package*. 2026, p. 14. URL: `https://mirrors.ctan.org/macros/latex/contrib/codeverbatim/doc/codeverbatim.pdf` (visited on 11/30/2026).

[4] Pablo González. *SCONTENTS - Stores LaTeX Contents*. 2024, p. 48. URL: `http://mirrors.ctan.org/macros/latex/contrib/scontents/scontents.pdf` (visited on 03/10/2025).

[5] Jobst Hoffmann. *The Listings Package*. 2024, p. 65. URL: `http://mirrors.ctan.org/macros/latex/contrib/listings/listings.pdf` (visited on 03/10/2025).

[6] Axel Sommerfeldt. *The newfloat package*. 2023, p. 8. URL: https://mirrors.ctan.org/macros/latex/contrib/newfloat/newfloat.pdf (visited on 11/30/2026).

[7] The LaTeX Team. *LaTeX for authors. current version*. 2026, p. 27. URL: https://mirrors.ctan.org/macros/latex/base/usrguide.pdf (visited on 08/07/2026).

[8] The LaTeX Team. *The LaTeX3 Interfaces. The LaTeX Project*. 2026, p. 385. URL: https://mirrors.ctan.org/macros/latex/required/l3kernel/interface3.pdf (visited on 08/07/2026).

[9] Timothy Van Zandt. *The 'fancyvrb' package. Fancy Verbatims in LATEX*. 2026, p. 24. URL: http://mirrors.ctan.org/macros/latex/contrib/fancyvrb/doc/fancyvrb-doc.pdf (visited on 06/19/2026).