A New Implementation of \LaTeX{}'s \verbbatim{} and \verbbatim*{} Environments.

Rainer Schöpf  
Zentrum für Datenverarbeitung  
der Universität Mainz  
Anselm-Frantz-von-Bentzel-Weg 12  
D-55099 Mainz  
Federal Republic of Germany  
Internet: Schoepf@Uni-Mainz.DE

Bernd Raichle  
Stettener Str. 73  
D-73732 Wäldenbronn  
Federal Republic of Germany  
Internet: raichle@azu.Informatik.Uni-Stuttgart.DE

Chris Rowley  
The Open University  
Parsifal College  
Finchley Road  
London NW3 7BG, England, UK  
Internet: C.A.Rowley@open.ac.uk

1999/12/14

Abstract

This package reimplements the \LaTeX{} \verbbatim{} and \verbbatim*{} environments. In addition it provides a \verbcomment{} environment that skips any commands or text between \verb\begin{comment}{} and the next \verb\end{comment}{}. It also defines the command \verbverbatiminput{} to input a whole file \verbverbatim{}.

1 Usage notes

\LaTeX{}'s \verbverbatim{} and \verbverbatim*{} environments have a few features that may give rise to problems. These are:
Verbatim style option

- Due to the method used to detect the closing \texttt{end[verbatim]} (i.e. macro parameter delimiting) you cannot leave spaces between the \texttt{end} token and \texttt{verbatim}.

- Since \TeX{} has to read all the text between the \texttt{begin[verbatim]} and the \texttt{end[verbatim]} before it can output anything, long verbatim listings may overflow \TeX{}'s memory.

Whereas the first of these points can be considered only a minor nuisance the other one is a real limitation.

This package file contains a reimplementation of the \texttt{verbatim} and \texttt{verbatim*} environments which overcomes these restrictions. There is, however, one incompatibility between the old and the new implementations of these environments: the old version would treat text on the same line as the \texttt{end[verbatim]} command as if it were on a line by itself.

This new version will simply ignore it.

(This is the price one has to pay for the removal of the old \texttt{verbatim} environment’s size limitations.) It will, however, issue a warning message of the form

\LaTeX{} warning: Characters dropped after \texttt{end[verbatim*]}!

This is not a real problem since this text can easily be put on the next line without affecting the output.

This new implementation also solves the second problem mentioned above: it is possible to leave spaces (but \textit{not} begin a new line) between the \texttt{end} and the \texttt{verbatim} or \texttt{verbatim*}:

\begin{verbatim*}
  test
test
\end{verbatim*}

Additionally we introduce a \texttt{comment} environment, with the effect that the text between \texttt{begin[comment]} and \texttt{end[comment]} is simply ignored, regardless of what it looks like. At first sight this seems to be quite different from the purpose of verbatim listing, but actually the implementation of these two concepts turns out to be very similar. Both rely on the fact that the text between \texttt{begin[...]} and \texttt{end[...]} is read by \TeX{} without interpreting any commands or special characters. The remaining difference between \texttt{verbatim} and \texttt{comment} is only that the text is to be typeset in the first case and to be thrown away in the latter. Note that these environments cannot be nested.

\verbatiminput{} is a command with one argument that inputs a file verbatim, i.e. the command \texttt{verbatiminput{xx.yy}} has the same effect as

\begin{verbatim}
  (Contents of the file \texttt{xx.yy})
\end{verbatim}

This command has also a \texttt{*}-variant that prints spaces as ~.
2 Interfaces for package writers

The verbatim environment of \LaTeX{} does not offer a good interface to programmers. In contrast, this package provides a simple mechanism to implement similar features, the comment environment implemented here being an example of what can be done and how.

2.1 Simple examples

It is now possible to use the verbatim environment to define environments of your own. E.g.,

\newenvironment{myverbatim}{\endgraf\noindent MYVERBATIM:%}{\endverbatim}

\begin{myverbatim}
test
test
\end{myverbatim}

Another way to use it is to write

\let\foo=\comment
\let\endfoo=\endcomment

and from that point on environment foo is the same as the comment environment, i.e. everything inside its body is ignored.

You may also add special commands after the verbatim macro is invoked, e.g.

\newenvironment{myverbatim}{\verbatim\myspecialverbatimsetup}{\endverbatim}

though you may want to learn about the hook \everyverbatim at this point.

However, there are still a number of restrictions:

1. You must not use the \begin or the \end command inside a definition, e.g. the following two examples will not work:

\newenvironment{myverbatim}{\endgraf\noindent MYVERBATIM:%}{\endverbatim}\endgraf\begin{verbatim}
\newenvironment{fred}{\begin{minipage}{30mm}\verbatim}{\endverbatim\end{minipage}}
If you try these examples, \TeX{} will report a “runaway argument” error. More generally, it is not possible to use \begin{...} or the related environments in the definition of the new environment. Instead, the correct way to define this environment would be

\begin{verbatim}
\newenvironment{fred}\
{\minipage{30mm}}\verbatim\
{\endverbatim\endminipage}
\end{verbatim}

2. You cannot use the verbatim environment inside user defined commands; e.g.,

\begin{verbatim}
\newcommand{\verbatimfile}{[1]}\
\begin{verbatim}input{#1}\end{verbatim}
\end{verbatim}
\end{verbatim}

does not work; nor does

\begin{verbatim}
\newcommand{\verbatimfile}{\verbatim input{#1} endverbatim}
\end{verbatim}

3. The name of the newly defined environment must not contain characters with category code other than 11 (letter) or 12 (other), or this will not work.

2.2 The interfaces

Let us start with the simple things. Sometimes it may be necessary to use a special typeface for your verbatim text, or perhaps the usual computer modern typewriter shape in a reduced size.

You may select this by redefining the macro \verbatim@font. This macro is executed at the beginning of every verbatim text to select the font shape. Do not use it for other purposes; if you find yourself abusing this you may want to read about the \every@verbatim hook below.

By default, \verbatim@font switches to the typewriter font and disables the ligatures contained therein.

There is a hook (i.e. a token register) called \every@verbatim whose contents are inserted into \TeX{}’s mouth just before every verbatim text. Please use the \addto@hook macro to add something to this hook. It is used as follows:

\begin{verbatim}
\addto@hook{\verbatim@font}{⟨commands to be added⟩}
\end{verbatim}

After all specific setup, like switching of category codes, has been done, the \verbatim@start macro is called. This starts the main loop of the scanning mechanism implemented here. Any other environment that wants to make use of this feature should execute this macro as its last action.

These are the things that concern the start of a verbatim environment. Once this (and other) setup has been done, the code in this package reads and processes characters from the input stream in the following way:
1. Before the first character of an input line is read, it executes the macro `\verbatim@startline`.

2. After some characters have been read, the macro `\verbatim@addtoline` is called with these characters as its only argument. This may happen several times per line (when an `\end` command is present on the line in question).

3. When the end of the line is reached, the macro `\verbatim@processline` is called to process the characters that `\verbatim@addtoline` has accumulated.

4. Finally, there is the macro `\verbatim@finish` that is called just before the environment is ended by a call to the `\end` macro.

To make this clear let us consider the standard `verbatim` environment. In this case the three macros above are defined as follows:

1. `\verbatim@startline` clears the character buffer (a token register).
2. `\verbatim@addtoline` adds its argument to the character buffer.
3. `\verbatim@processline` typesets the characters accumulated in the buffer.

With this it is very simple to implement the `comment` environment: in this case `\verbatim@startline` and `\verbatim@processline` are defined to be no-ops whereas `\verbatim@addtoline` discards its argument.

Let’s use this to define a variant of the `verbatim` environment that prints line numbers in the left margin. Assume that this would be done by a counter called `VerbatimLineNo`. Assuming that this counter was initialized properly by the environment, `\verbatim@processline` would be defined in this case as

```latex\def\verbatim@processline{\addtocounter{VerbatimLineNo}{1}\leavevmode\llap{\theVerbatimLineNo\hskip\@totalleftmargin}\the\verbatim@line\par}
```

A further possibility is to define a variant of the `verbatim` environment that boxes and centers the whole verbatim text. Note that the boxed text should be less than a page otherwise you have to change this example.

```latex\def\verbatimboxed#1{\begingroup\verbatiminput{#1}}\endgroup\begin{center}\fbox{\verbatimboxed{#1}}\end{center}
```
As a final nontrivial example we describe the definition of an environment called `verbatimwrite`. It writes all text in its body to a file whose name is given as an argument. We assume that a stream number called `verbatim@out` has already been reserved by means of the `newwrite` macro.

Let’s begin with the definition of the macro `verbatimwrite`.

\def\verbatimwrite#1{%  
First we call `@bsphack` so that this environment does not influence the spacing. Then we open the file and set the category codes of all special characters:

```
\@bsphack
\immediate\openout \verbatim@out #1
\let\do\@makeother\dospecials
\catcode'\^^M\active
```

The default definitions of the macros

```
verbatim@startline
verbatim@addtoline
verbatim@finish
```

are also used in this environment. Only the macro `verbatim@processline` has to be changed before `verbatim@start` is called:

```
\def\verbatim@processline{%
  \immediate\write\verbatim@out{\theverbatim@line}%
  \verbatim@start}
```

The definition of `endverbatimwrite` is very simple: we close the stream and call `@esphack` to get the spacing right.

```
\def\endverbatimwrite{\immediate\closeout\verbatim@out\@esphack}
```

## 3 The implementation

The very first thing we do is to ensure that this file is not read in twice. To this end we check whether the macro `verbatim@@@` is defined. If so, we just stop reading this file. The ‘package’ guard here allows most of the code to be excluded when extracting the driver file for testing this package.

```
1 ⟨∗package⟩
2 ∃NeedsTeXFormat{LaTeX2e}
3 ∃ProvidesPackage{verbatim}
4  [2000/01/07 v1.5m LaTeX2e package for verbatim enhancements]
5 ∃Ifundefined{verbatim@@@}{}{\endinput}
```

We use a mechanism similar to the one implemented for the `comment`…`endcomment` macro in `AMS-\TeX`: We input one line at a time and check if it contains the `\end{...}` tokens. Then we can decide whether we have reached the end of the verbatim text, or must continue.
3.1 Preliminaries

\everyverbatim

The hook (i.e. token register) \everyverbatim is initialized to \textit{(empty)}.

6 \newtoks\everyverbatim
7 \everyverbatim={}

@makeother

@makeother takes as argument a character and changes its category code to 12 (other).

8 \def\@makeother#1{\catcode'#112\relax}

@vobeyspaces

The macro \@vobeyspaces causes spaces in the input to be printed as spaces in the output.

9 \begingroup
10 \catcode'\ =\active%
11 \def\x[{\def\@vobeyspaces{\catcode'\ active\let \@xobeysp}}
12 \expandafter\endgroup\x

@xobeysp

The macro \@xobeysp produces exactly one space in the output, protected against breaking just before it. (\@M is an abbreviation for the number 10000.)

13 \def\@xobeysp{\leavevmode\penalty\@M\ }

verbatim@line

We use a newly defined token register called \verbatim@line that will be used as the character buffer.

14 \newtoks\verbatim@line

The following four macros are defined globally in a way suitable for the \verbatim and \verbatim* environments.

verbatim@startline

verbatim@startline initializes processing of a line by emptying the character buffer (verbatim@line).

15 \def\verbatim@startline{\verbatim@line{}}

verbatim@addtoline

verbatim@addtoline adds the tokens in its argument to our buffer register \verbatim@line without expanding them.

16 \def\verbatim@addtoline#1{\verbatim@line\expandafter{\the\verbatim@line#1}}

Processing a line inside a \verbatim or \verbatim* environment means printing it. Ending the line means that we have to begin a new paragraph. We use \par for this purpose. Note that \par is redefined in \verbatim to force \TeX{} into horizontal mode and to insert an empty box so that empty lines in the input do appear in the output.

18 \def\verbatim@processline{\the\verbatim@line\par}

verbatim@finish

As a default, \verbatim@finish processes the remaining characters. When this macro is called we are facing the following problem: when the \verbatim@finish command is encountered \verbatim@processline is called to process the characters preceding the command on the same line. If there are none, an empty line would be output if we did not check for this case.
If the line is empty \texttt{\verbatim@line} expands to nothing. To test this we use a trick similar to that on p. 376 of the \TeX{}book, but with $\ldots$ instead of the ! tokens. These $\ldots$ tokens can never have the same category code as a $\ldots$ token that might possibly appear in the token register \texttt{\verbatim@line}, as such a token will always have been read with category code 12 (other). Note that \texttt{\ifcat} expands the following tokens so that \texttt{\the\verbatim@line} is replaced by the accumulated characters

\begin{verbatim}
\def\verbatim@finish{\ifcat$\the\verbatim@line$\else\verbatim@processline\fi}
\end{verbatim}

3.2 The \texttt{verbatim} and \texttt{verbatim*} environments

\begin{verbatim}
\verbatim@font
\end{verbatim}

\texttt{\verbatim@font} We start by defining the macro \texttt{\verbatim@font} that is to select the font and to set font-dependent parameters. Then we go through \texttt{\verbatim@nolig@list} to avoid certain ligatures. \texttt{\verbatim@nolig@list} is a macro defined in the \LaTeX{} kernel to expand to

\begin{verbatim}
\do\do\do\do\do\do\do-
\end{verbatim}

All the characters in this list can be part of a ligature in some font or other.

\begin{verbatim}
\def\verbatim@font{\normalfont\ttfamily
  \hyphenchar\font\m@ne
  \let\do\do@noligs
  \verbatim@nolig@list}
\end{verbatim}

\begin{verbatim}
\@verbatim
\end{verbatim}

The macro \texttt{@verbatim} sets up things properly. First of all, the tokens of the \texttt{\everyverbatim} hook are inserted. Then a \texttt{trivlist} environment is started and its first \texttt{item} command inserted. Each line of the \texttt{verbatim} or \texttt{verbatim*} environment will be treated as a separate paragraph.

\begin{verbatim}
\def\verbatim\{\the\everyverbatim
\trivlist \item \relax
\end{verbatim}

The following extra vertical space is for compatibility with the \LaTeX{} kernel: otherwise, using the \texttt{verbatim} package changes the vertical spacing of a \texttt{verbatim} environment nested within a \texttt{quote} environment.

\begin{verbatim}
\if@minipage\else\vskip\parskip\fi
\end{verbatim}

The paragraph parameters are set appropriately: the penalty at the beginning of the environment, left and right margins, paragraph indentation, the glue to fill the last line, and the vertical space between paragraphs. The latter space has to be zero since we do not want to add extra space between lines.

\begin{verbatim}
\@beginparpenalty \predisplaypenalty
\leftskip\@totalleftmargin\rightskip\z@ \parindent\z@ \parfillskip\@flushglue \parskip\z@
\end{verbatim}

There's one point to make here: the \texttt{list} environment uses \TeX{}'s \texttt{parshape} primitive to get a special indentation for the first line of the list. If the list begins with a \texttt{verbatim} environment this \texttt{parshape} is still in effect. Therefore we have to reset this internal parameter explicitly. We could do this by assigning 0 to
\parshape. However, there is a simpler way to achieve this: we simply tell \TeX to start a new paragraph. As is explained on p. 103 of the \TeX book, this resets \parshape to zero.

We now ensure that \par has the correct definition, namely to force \TeX into horizontal mode and to include an empty box. This is to ensure that empty lines do appear in the output. Afterwards, we insert the \interlinepenalty since \TeX does not add a penalty between paragraphs (here: lines) by its own initiative. Otherwise a verbatim environment could be broken across pages even if a samepage declaration were present.

However, in a top-aligned minipage, this will result in an extra empty line added at the top. Therefore, a slightly more complicated construct is necessary. One of the important things here is the inclusion of \leavevmode as the first macro in the first line, for example, a blank verbatim line is the first thing in a list item.

\verbatim* Now we define the toplevel macros. \verbatim is slightly changed: after setting up things properly it calls \verbatim@start. This is done inside a group, so that \verbatim can be used directly, without \begin.

\verbatim* is defined accordingly.
To end the `verbatim` and `verbatim*` environments it is only necessary to finish the `trivlist` environment started in `@verbatim` and close the corresponding group.

\[ \text{\verbatim*} \]

\[
\text{\def\endverbatim}{\endtrivlist\endgroup}\]
\[
\expandafter\let\csname endverbatim*\endcsname =\endverbatim\]

3.3 The comment environment

The `\comment` macro is similar to `\verbatim*`. However, we do not need to switch fonts or set special formatting parameters such as `\parindent` or `\parskip`. We need only set the category code of all special characters to 12 (other) and that of `\^M` (the end of line character) to 13 (active). The latter is needed for macro parameter delimiter matching in the internal macros defined below. In contrast to the default definitions used by the `\verbatim` and `\verbatim*` macros, we define `\verbatim@addtoline` to throw away its argument and `\verbatim@processline`, `\verbatim@startline`, and `\verbatim@finish` to act as no-ops. Then we call `\verbatim@`. But the first thing we do is to call `\@bsphack` so that this environment has no influence whatsoever upon the spacing.

\[
\text{\def\comment}{\@bsphack}\]
\[
\let\do\@makeother\dospecials\catcode'\^M\active\]
\[
\let\verbatim@startline\relax\]
\[
\let\verbatim@addtoline\@gobble\]
\[
\let\verbatim@processline\relax\]
\[
\let\verbatim@finish\relax\]
\[
\verbatim@\]
\[
\endcomment\]

\[
\text{\def\endcomment}{\@esphack}\]

This environment has no influence whatsoever upon the spacing. The `\end` macro closes the group and therefore takes care of restoring everything we changed.

\[
\let\endcomment=\@esphack\]

3.4 The main loop

Here comes the tricky part: During the definition of the macros we need to use the special characters `\`, `{`, and `}` not only with their normal category codes, but also with category code 12 (other). We achieve this by the following trick: first we tell \TeX that `\`, `{`, and `}` are the lowercase versions of `!`, `[`, and `]`. Then we replace every occurrence of `\`, `{`, and `}` that should be read with category code 12 by `!`, `[`, and `]`, respectively, and give the whole list of tokens to `\lowercase`, knowing that category codes are not altered by this primitive!

But first we have ensure that `!`, `[`, and `]` themselves have the correct category code! To allow special settings of these codes we hide their setting in the macro `\verb@catcodes`. If it is already defined our new definition is skipped.

\[
\@ifundefined{vrb@catcodes}%\]
\[
\{\def\verb@catcodes{%\]
\[
\catcode'\!'12\catcode'\[12\catcode'\]12\}\}%\]
This trick allows us to use this code for applications where other category codes are in effect.

We start a group to keep the category code changes local.

```
\begingroup
\verb@catcodes
\lccode'\!='\ \lccode'\[='\{ \lccode'\]=]='\}
```

We also need the end-of-line character `\^M`, as an active character. If we were to simply write `\catcode'\^M=\active` then we would get an unwanted active end of line character at the end of every line of the following macro definitions. Therefore we use the same trick as above: we write a tilde `~` instead of `\^M` and pretend that the latter is the lowercase variant of the former. Thus we have to ensure now that the tilde character has category code 13 (active).

```
\catcode'~=\active \lccode'~='\^M
```

The use of the `\lowercase` primitive leads to one problem: the uppercase character `\c` needs to be used in the code below and its case must be preserved. So we add the command:

```
\lccode'\C=\C
```

Now we start the token list passed to `\lowercase`. We use the following little trick (proposed by Bernd Raichle): The very first token in the token list we give to `\lowercase` is the `\endgroup` primitive. This means that it is processed by TeX immediately after `\lowercase` has finished its operation, thus ending the group started by `\begingroup` above. This avoids the global definition of all macros.

```
\lowercase{\endgroup}
```

The purpose of `\verbatim@start` is to check whether there are any characters on the same line as the `\begin{verbatim}` and to pretend that they were on a line by themselves. On the other hand, if there are no characters remaining on the current line we shall just find an end of line character. `\verbatim@start` performs its task by first grabbing the following character (its argument). This argument is then compared to an active `\^M`, the end of line character.

```
\def\verbatim@start#1{% 
\verbatim@startline 
\if\noexpand#1\noexpand~% 
\else \def\next{\verbatim@#1}\fi 
}
```

If this is true we transfer control to `\verbatim@` to process the next line. We use `\next` as the macro which will continue the work.

```
\let\next\verbatim@
```

Otherwise, we define `\next` to expand to a call to `\verbatim@` followed by the character just read so that it is reinserted into the text. This means that those characters remaining on this line are handled as if they formed a line by themselves.

```
\else \def\next{\verbatim@#1}\fi
```

Finally we call `\next`.

```
\verbatim@start
\verbatim@startline
\verbatim@start
\verbatim@start
```

---

**Verbatim style option**

This trick allows us to use this code for applications where other category codes are in effect.

We start a group to keep the category code changes local.

```
\begingroup
\verb@catcodes
\lccode'\!='\ \lccode'\[='\{ \lccode'\]=]='\}
```

We also need the end-of-line character `\^M`, as an active character. If we were to simply write `\catcode'\^M=\active` then we would get an unwanted active end of line character at the end of every line of the following macro definitions. Therefore we use the same trick as above: we write a tilde `~` instead of `\^M` and pretend that the latter is the lowercase variant of the former. Thus we have to ensure now that the tilde character has category code 13 (active).

```
\catcode'~=\active \lccode'~='\^M
```

The use of the `\lowercase` primitive leads to one problem: the uppercase character `\c` needs to be used in the code below and its case must be preserved. So we add the command:

```
\lccode'\C=\C
```

Now we start the token list passed to `\lowercase`. We use the following little trick (proposed by Bernd Raichle): The very first token in the token list we give to `\lowercase` is the `\endgroup` primitive. This means that it is processed by TeX immediately after `\lowercase` has finished its operation, thus ending the group started by `\begingroup` above. This avoids the global definition of all macros.

```
\lowercase{\endgroup}
```

The purpose of `\verbatim@start` is to check whether there are any characters on the same line as the `\begin{verbatim}` and to pretend that they were on a line by themselves. On the other hand, if there are no characters remaining on the current line we shall just find an end of line character. `\verbatim@start` performs its task by first grabbing the following character (its argument). This argument is then compared to an active `\^M`, the end of line character.

```
\def\verbatim@start#1{% 
\verbatim@startline 
\if\noexpand#1\noexpand~% 
\else \def\next{\verbatim@#1}\fi 
}
```

If this is true we transfer control to `\verbatim@` to process the next line. We use `\next` as the macro which will continue the work.

```
\let\next\verbatim@
```

Otherwise, we define `\next` to expand to a call to `\verbatim@` followed by the character just read so that it is reinserted into the text. This means that those characters remaining on this line are handled as if they formed a line by themselves.

```
\else \def\next{\verbatim@#1}\fi
```

Finally we call `\next`.
The three macros `\verbatim@`, `\verbatim@@`, and `\verbatim@@@` form the “main loop” of the `verbatim` environment. The purpose of `\verbatim@` is to read exactly one line of input. `\verbatim@@` and `\verbatim@@@` work together to find out whether the four characters `\end` (all with category code 12 (other)) occur in that line. If so, `\verbatim@@@` will call `\verbatim@test` to check whether this `\end` is part of `\end\verbatim` and will terminate the environment if this is the case. Otherwise we continue as if nothing had happened. So let’s have a look at the definition of `\verbatim@`:

```latex
\def\verbatim@#1~{\verbatim@@#1!end\@nil}\
```

Note that the `!` character will have been replaced by a `\` with category code 12 (other) by the `\lowercase` primitive governing this code before the definition of this macro actually takes place. That means that it takes the line, puts `\end` (four character tokens) and `\@nil` (one control sequence token) as a delimiter behind it, and then calls `\verbatim@@`.

`\verbatim@@` takes everything up to the next occurrence of the four characters `\end` as its argument.

```latex
\def\verbatim@@#1!end{\
```

That means: if they do not occur in the original line, then argument `#1` is the whole input line, and `\@nil` is the next token to be processed. However, if the four characters `\end` are part of the original line, then `#1` consists of the characters in front of `\end`, and the next token is the following character (always remember that the line was lengthened by five tokens). Whatever `#1` may be, it is verbatim text, so `#1` is added to the line currently built.

```latex
\verbatim@addtoline{#1}\
```

The next token in the input stream is of special interest to us. Therefore `\futurelet` defines `\next` to be equal to it before calling `\verbatim@@@`.

```latex
\futurelet\next\verbatim@@@\
```

`\verbatim@@@` will now read the rest of the tokens on the current line, up to the final `\@nil` token.

```latex
\def\verbatim@@@#1\@nil{\
```

If the first of the above two cases occurred, i.e. no `\end` characters were on that line, `#1` is empty and `\next` is equal to `\@nil`. This is easily checked.

```latex
\ifx\next\@nil\
```

If so, this was a simple line. We finish it by processing the line we accumulated so far. Then we prepare to read the next line.

```latex
\verbatim@processline
\verbatim@startline
\let\next\verbatim@
```

Otherwise we have to check what follows these `\end` tokens.
Before we continue, it’s a good idea to stop for a moment and remember where we are: We have just read the four character tokens \end and must now check whether the name of the environment (surrounded by braces) follows. To this end we define a macro called \@tempa that reads exactly one character and decides what to do next. This macro should do the following: skip spaces until it encounters either a left brace or the end of the line. But it is important to remember which characters are skipped. The \end⟨optional spaces⟩ characters may be part of the verbatim text, i.e. these characters must be printed.

Assume for example that the current line contains
\end{AVeryLongEnvironmentName}
As we shall soon see, the scanning mechanism implemented here will not find out that this is text to be printed until it has read the right brace. Therefore we need a way to accumulate the characters read so that we can reinsert them if necessary. The token register \@temptokena is used for this purpose.

Before we do this we have to get rid of the superfluous \end tokens at the end of the line. To this end we define a temporary macro whose argument is delimited by \end\@nil (four character tokens and one control sequence token) to be used below on the rest of the line, after appending a \@nil token to it. (Note that this token can never appear in \#1.) We use the following definition of \@tempa to get the rest of the line (after the first \end).

\def\@tempa##1!end\@nil{##1}\

We mentioned already that we use token register \@temptokena to remember the characters we skip, in case we need them again. We initialize this with the \end we have thrown away in the call to \@tempa.

\@temptokena{!end}\

We shall now call \verbatim@test to process the characters remaining on the current line. But wait a moment: we cannot simply call this macro since we have already read the whole line. Therefore we have to first expand the macro \@tempa to insert them again after the \verbatim@test token. A \^M character is appended to denote the end of the line. (Remember that this character comes disguised as a tilde.)

\def\next{\expandafter{verbatim@test\@tempa\#1\@nil~}}\

That’s almost all, but we still have to now call \next to do the work.

\verbatim@test We define \verbatim@test to investigate every token in turn.

\def\verbatim@test\#1{%
First of all we set \next equal to \verbatim@test in case this macro must call itself recursively in order to skip spaces.

\let\next\verbatim@test

We have to distinguish four cases:
1. The next token is a `\^-M`, i.e. we reached the end of the line. That means that nothing special was found. Note that we use `\if` for the following comparisons so that the category code of the characters is irrelevant.

```
\if\noexpand\equal\noexpand^-M
```

We add the characters accumulated in token register `\@temptokena` to the current line. Since `\verbatim@addtoline` does not expand its argument, we have to do the expansion at this point. Then we `\let \next` equal to `\verbatim@` to prepare to read the next line.

```
\expandafter\verbatim@addtoline
\expandafter{\the\@temptokena}\
\verbatim@processline
\verbatim@startline
\let\next\verbatim@\%
```

2. A space character follows. This is allowed, so we add it to `\@temptokena` and continue.

```
\else \if\noexpand\equal\%
\@temptokena\expandafter{\the\@ temptokena}\%
```

3. An open brace follows. This is the most interesting case. We must now collect characters until we read the closing brace and check whether they form the environment name. This will be done by `\verbatim@testend`, so here we let `\next` equal this macro. Again we will process the rest of the line, character by character. The characters forming the name of the environment will be accumulated in `\@tempc`. We initialize this macro to expand to nothing.

```
\else \if\noexpand\equal\%
\let\@tempc\@empty
\let\next\verbatim@testend
```

Note that the `[` character will be a `{` when this macro is defined.

4. Any other character means that the `\end` was part of the verbatim text. Add the characters to the current line and prepare to call `\verbatim@` to process the rest of the line.

```
\else
\expandafter\verbatim@addtoline
\expandafter{\the\@temptokena}\
\def\next{\verbatim@\%
\fi\fi\fi
```

The last thing this macro does is to call `\next` to continue processing.

```
\next\%
```
\verbatim@testend \verbatim@testend is called when \end\{optional spaces\}\{ was seen. Its task is to
scan everything up to the next } and to call \verbatim@testend. If no } is found
it must reinsert the characters it read and return to \verbatim@. The following
definition is similar to that of \verbatim@test: it takes the next character and
decides what to do.

\def\verbatim@testend#1{% Again, we have four cases:

1. ^\M{}: As no } is found in the current line, add the characters to the buffer. To
avoid a complicated construction for expanding \@emptokena and \@emptpc
we do it in two steps. Then we continue with \verbatim@ to process the
next line.

\if\noexpand#1\noexpand\% \verbatim@addtoline
\verbatim@addtoline\verbatim@addtoline\verbatim@addtoline\verbatim@addtoline
\verbatim@addtoline\verbatim@addtoline\verbatim@addtoline\verbatim@addtoline
\verbatim@addtoline\verbatim@addtoline\verbatim@addtoline\verbatim@addtoline
\verbatim@addtoline\verbatim@addtoline\verbatim@addtoline
\verbatim@startline \let\next\verbatim@startline
\verbatim@testend

2. \}: Call \verbatim@testend to check if this is the right environment name.

\else\if\noexpand#1\noexpand\%
\verbatim@testend

3. \: This character must not occur in the name of an environment. Thus we
stop collecting characters. In principle, the same argument would apply to
other characters as well, e.g., {. However, \ is a special case, since it may
be the first character of \end. This means that we have to look again for
\end\{\{environment name\}\}. Note that we prefixed the ! by a \noexpand
primitive, to protect ourselves against it being an active character.

\else\if\noexpand#1\noexpand!\%
\verbatim@testend

4. Any other character: collect it and continue. We cannot use \edef to define
\emptpc since its replacement text might contain active character tokens.

\else \verbatim@def\verbatim@expandafter\verbatim@expandafter\verbatim@expandafter\verbatim@expandafter\verbatim@expandafter
\verbatim@expandafter\verbatim@expandafter\verbatim@expandafter\verbatim@expandafter\verbatim@expandafter\verbatim@expandafter\verbatim@expandafter\verbatim@expandafter\verbatim@expandafter\verbatim@expandafter\verbatim@expandafter\verbatim@expandafter}
\verbatim@testend\verbatim@testend

As before, the macro ends by calling itself, to process the next character if appro-
priate.

\next\%
Verbatim style option

```
\verbatim@@testend
```

Unlike the previous macros `\verbatim@@testend` is simple: it has only to check if the `\end{...}` matches the corresponding `\begin{...}`.

125 \def\verbatim@@testend{%

We use `\next` again to define the things that are to be done. Remember that the name of the current environment is held in `\@currenvir`, the characters accumulated by `\verbatim@addtoline` are in `\@tempc`. So we simply compare these and prepare to execute `\end{⟨current environment⟩}` macro if they match. Before we do this we call `\verbatim@finish` to process the last line. We define `\next` via `\edef` so that `\@currenvir` is replaced by its expansion. Therefore we need `\noexpand` to inhibit the expansion of `\end` at this point.

126 \ifx\@tempc\@currenvir
127 \verbatim@addtoline
128 \edef\next{\noexpand\verbatim@addtoline{\@currenvir}}%

Without this trick the `\end` command would not be able to correctly check whether its argument matches the name of the current environment and you’d get an interesting \TeX error message such as:

```
! \begin{verbatim*} ended by \end{verbatim*}.
```

But what do we do with the rest of the characters, those that remain on that line? We call `\verbatim@rescan` to take care of that. Its first argument is the name of the environment just ended, in case we need it again. `\verbatim@rescan` takes the list of characters to be reprocessed as its second argument. (This token list was inserted after the current macro by `\verbatim@@`). Since we are still in an `\edef` we protect it by means of `\noexpand`.

129 \noexpand\verbatim@rescan{\@currenvir}%

If the names do not match, we reinsert everything read up to now and prepare to call `\verbatim@addtoline` to process the rest of the line.

130 \else
131 \expandafter{\verbatim@addtoline
132 \expandafter{\the@temptokena}%
133 \expandafter{\verbatim@addtoline
134 \expandafter{\@tempc}}%
135 \let\next\verbatim@
136 \fi

Finally we call `\next`.

```
\verbatim@rescan
```

In principle `\verbatim@rescan` could be used to analyse the characters remaining after the `\end{...}` command and pretend that these were read “properly”, assuming “standard” category codes are in force.¹ But this is not always possible (when there are unmatched curly braces in the rest of the line). Besides, we think that this is not worth the effort: After a `verbatim` or `verbatim*` environment a new line in the output is begun anyway, and an `\end{comment}` can easily be

¹Remember that they were all read with category codes 11 (letter) and 12 (other) so that control sequences are not recognized as such.
Verbatim style option

put on a line by itself. So there is no reason why there should be any text here. For the benefit of the user who did put something there (a comment, perhaps) we simply issue a warning and drop them. The method of testing is explained in Appendix D, p. 376 of the \TeX\book. We use "\M" instead of the ! character used there since this is a character that cannot appear in #1. The two \noexpand primitives are necessary to avoid expansion of active characters and macros.

One extra subtlety should be noted here: remember that the token list we are currently building will first be processed by the \lowercase primitive before \TeX\ carries out the definitions. This means that the 'C' character in the argument to the \Warning macro must be protected against being changed to 'c'. That's the reason why we added the \lccode'C='C assignment above. We can now finish the argument to \lowercase as well as the group in which the category codes were changed.

\verbatim@rescan#1#2~{\if\noexpand\noexpand#2~\else
\verbatim@finish\string\end{#1}\fi}

3.5 The \verbatiminput command

\verbatim@startline

When the file is not found we issue a warning.

\verbatim@readfile

\verbatim@finish

At this point we pass the name of the file to \addtofilelist so that its appearance appears in the output of a \listfiles command. In addition, we use \ProvidesFile to make a log entry in the transcript file and to distinguish files read in via \verbatiminput from others.

\verbatim@read@file

While reading from the file it is useful to switch off the recognition of the end-of-line character. This saves us stripping off spaces from the contents of the line.

\verbatim@open

\verbatim@close
All the work is done in \verbatim@read@file. It reads the input file line by line and recursively calls itself until the end of the file.

For each line we call \verbatim@addtoline with the contents of the line. \verbatim@processline is called next.

After processing the line we call \verbatim@startline to initialize all before we read the next line.

Without \verbatim@read@file each call of \verbatim@read@file uses space in \TeX's input stack.\footnote{A standard \TeX would report an overflow error if you try to read a file with more than ca. 200 lines. The same error occurs if the first line of code in §390 of “\TeX: The Program” is missing.}

\verbatiminput first starts a group to keep font and category changes local. Then it calls the macro \verbatiminput with additional arguments, depending on whether an asterisk follows.

Then it reads in the file, finishes off the \texttt{trivlist} environment started by \verbatim and closes the group. This restores everything to its normal settings.

If the file is not found a more or less helpful message is printed. The final \texttt{endgroup} is needed to close the group started in \verbatiminput above.
3.6 Getting verbatim text into arguments.

One way of achieving this is to define a macro (command) whose expansion is the required verbatim text. This command can then be used anywhere that the verbatim text is required. It can be used in arguments, even moving ones, but it is fragile (at least, the version here is).

Here is some code which claims to provide this. It is a much revised version of something I (Chris) did about 2 years ago. Maybe it needs further revision.

It is only intended as an extension to \verb, not to the \verbatim environment. It should therefore, perhaps, treat line-ends similarly to whatever is best for \verb.

\verbtext
This is the command to produce a new macro whose expansion is verbatim text. This command itself cannot be used in arguments, of course! It is used as follows:

\verbtext{\myverb}^%{ &~_\}}@ #"

The rules for delimiting the verbatim text are the same as those for \verb.

I think that a temporary switch is safe here: if not, then suitable \let's can be used.

\verbtext #1#2{% \beggingroup \let\do@makeother \dospecials \let\do@noligs \verbatim@nolig@list \@vobeyspaces \catcode'#2\active \catcode'~\active \lccode'~'#2% \lowercase

We use a temporary macro here and a trick so that the definition of the command itself can be done inside the group and be a local definition (there may be better ways to achieve this).

\verbtext #1#1-% \whitespaces

If these \noexpand's were \noexpand\protect\noexpand, would this make these things robust?

\verbtext #1{\noexpand \@verbatimmcheck
\bgroup
\if@tempswa
\noexpand \@visiblespaces
\fi
\noexpand \@verbatimsetup
##1%
Verbatim style option

\egroup}\%
\}
This sets up the correct type of group for the mode: it must not be expanded at define time!
\def \@verbtextmcheck {\%
\relax\ifmmode\hbox\else\leavevmode\null\fi\}
This contains other things which should not be expanded during the definition.
\def \@verbtextsetup {\frenchspacing\verbatim@font\verbtextstyle\}
The command \verbtextstyle is a document-level hook which can be used to override the predefined typographic treatment of commands defined with \newverbtext commands.
\visiblespaces and \whitespaces are examples of possible values of this hook.
\let \verbtextstyle \relax
\def \visiblespaces {\chardef \ 32\relax}
\def \whitespaces {\let \ @@space}
\let @@space \ %
\verbtext

4 Testing the implementation

For testing the implementation and for demonstration we provide an extra file. It can be extracted by using the conditional ‘testdriver’. It uses a small input file called ‘verbatim.tst’ that is distributed separately. Again, we use individual ‘+’ guards.
(*testdriver)
\documentclass{article}
\usepackage{verbatim}
\newenvironment{myverbatim}{\endgraf\noindent MYVERBATIM:\verbatim}{\endverbatim}
Verbatim style option

\makeatletter

newenvironment{verbatimlisting}[1]%
{\def\verbatim@startline{\input{#1}%
\def\verbatim@startline{\verbatim@line{}}%\verbatim@startline}%\verbatim}{\endverbatim}

\newwrite\verbatim@out

\newenvironment{verbatimwrite}[1]%
{\@bsphack\immediate\openout \verbatim@out #1\let\do\@makeother\dospecials\catcode'\^^M\active\def\verbatim@processline{\immediate\write\verbatim@out{\the\verbatim@line}}%\verbatim@start}%
{\immediate\closeout\verbatim@out\@esphack}

\makeatother

\addtolength{\textwidth}{30pt}
\begin{document}
\typeout{}\typeout{===> Expect 'characters dropped' warning messages in this test! <===} \typeout{}
\begin{verbatim}
test
\end{verbatim} Further text to be typeset: $\alpha$.
\end{verbatim}

\begin{verbatim}
test
\end{verbatim*}

\begin{verbatim}
test
\end{verbatim}

\begin{verbatim}
test
\end{verbatim} Further text to be typeset: $\alpha$.
First of Chris Rowley’s fiendish tests:
\begin{verbatim}
the double end test<text>
\end{verbatim} or even \end \end{verbatim}
%
%not \end\ended??
%\end{verbatim}

Another of Chris’ devils:
\begin{verbatim}
the single brace test<text>
\end{not the end\end{verbatim}
%
%not \end\ended??
%\end{verbatim}

Back to my own tests:
\begin{myverbatim}
test
test
\end {myverbatim} rest of line

Test of empty verbatim:
\begin{verbatim}
\end{verbatim}

Additional verbatim text
...
And here for listing a file:
\verbatiminput{verbtest.tex}
And again, with explicit spaces:
\verbatiminput*{verbtest.tex}
Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text
\begin{comment}
test
\end{verbatim*}
test\end{comment} Further text to be typeset: $\alpha$.
Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text
\begin{comment} bla bla
Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text
\end{comment}
\begin{verbatimwrite}{verbtest.txt}
asfa<fa<df
sdfsdfasd
asdfa<fsa
\end{verbatimwrite}