CHAPTER 3

Basic Formatting Tools

The way information is presented visually can influence, to a large extent, the message as it is understood by the reader. Therefore, it is important that you use the best possible tools available to convey the precise meaning of your words. It must, however, be emphasized that visual presentation forms should aid the reader in understanding the text, and should not distract his or her attention. For this reason, visual consistency and uniform conventions for the visual clues are a must, and the way given structural elements are highlighted should be the same throughout a document. This constraint is most easily implemented by defining a specific command or environment for each document element that has to be treated specially and by grouping these commands and environments in a package file or in the document preamble. By using exclusively these commands, you can be sure of a consistent presentation form.

This chapter explains various ways for highlighting parts of a document. The first part looks at how short text fragments or paragraphs can be made to stand out and describes tools to manipulate such elements.

The second part deals with the different kind of "notes", such as footnotes, marginal notes, and endnotes, and explains how they can be customized to conform to different styles, if necessary.

Typesetting lists is the subject of the third part. First, the various parameters and commands controlling the standard LaTeX lists, `enumerate`, `itemize`, and `description`, are discussed. Then, the extensions provided by the paralist package and the concept of "headed lists" exemplified by the amsthm package are presented. These will probably satisfy the structure and layout requirements of most readers. If not, then the remainder of this part introduces the generic `list` environment and explains how to build custom layouts by varying the values of the parameters controlling it.

The fourth part explains how to simulate “verbatim” text. In particular, we have a detailed look at the powerful packages fancyvrb and listings.

The final part presents packages that deal with line numbering, handling of columns, such as parallel text in two columns, or solving the problem of producing multiple columns.

3.1 Phrases and paragraphs

In this section we deal with small text fragments and explain how they can be manipulated and highlighted in a consistent manner by giving them a visual appearance different from the one used for the main text.

We start by discussing how to define commands that take care of the space after them, then show a way to produce professional-looking marks of omission.

For highlighting text you can customize the font shape, weight, or size (see Section 7.3.1 on page 338). Text can also be underlined, or the spacing between letters can be varied. Ways for performing such operations are offered by the four packages relsize, textcase, ulem, and soul.

The remainder of this section then turns to paragraph-related issues, such as producing large initial letters at the start of a paragraph, modifying paragraph justification, altering the vertical spacing between lines of a paragraph, and introducing rectangular holes into it, that can be filled with small pictures, among other things.

3.1.1 xspace—Gentle spacing after a macro

The small package xspace (by David Carlisle) defines the `\xspace` command, for use at the end of macros that produce text. It adds a space unless the macro is followed by certain punctuation characters.

The `\xspace` command saves you from having to type `\␣` or `{}` after most occurrences of a macro name in text. However, if either of these constructs follows `\xspace`, a space is not added by `\xspace`. This means that it is safe to add `\xspace` to the end of an existing macro without making too many changes in your document. Possible candidates for `\xspace` are commands for abbreviations such as “e.g.,” and “i.e.,”.

```
\newcommand\eg{e.g.,\xspace}
\newcommand\ie{i.e.,\xspace}
\newcommand\etc{etc.\@\xspace}
```

Notice the use of the `\@` command to generate the correct kind of space. If used to the right of a punctuation character, it prevents extra space from being added: the dot will not be regarded as an end-of-sentence symbol. Using it on the left forces LaTeX to interpret the dot as an end-of-sentence symbol.

Sometimes `\xspace` may make a wrong decision and add a space when it is not required. In such cases, follow the macro with `{}`, which will suppress this space.

Great Britain was unified in 1707.
Great Britain, the United States of America, and Canada have close cultural links.

3-1-1

```
\usepackage{xspace}
\newcommand\USA{United States of America\xspace}
\newcommand\GB {Great Britain\xspace}
\GB was unified in 1707.\\ \GB, the \USA, and
Canada have close cultural links.
```

3.1.2 `ellipsis`, `lips`—Marks of omission

Omission marks are universally represented by three consecutive periods (also known as an *ellipsis*). Their spacing, however, depends on house style and typographic conventions, and significant difference are observed. In French, according to Hart [63] or *The Chicago Manual of Style* [38], “points de suspension” are set close together and immediately follow the preceding word with a space on the right:

C’est une chose... bien difficile.

In German, according to the Duden [44], “Auslassungspunkte” have space on the left *and* right unless they mark missing letters within a word or a punctuation after them is kept:

Du E... du! Scher dich zum ...!

Elsewhere, such as in British and American typography, the dots are sometimes set with full word spaces between them and rather complex rules determine how to handle other punctuation marks at either end.

LaTeX offers the commands `\dots` and `\textellipsis` to produce closely spaced omission marks. Unfortunately, the standard definition (inherited from plain TeX) produces uneven spacing at the left and right—unsuitable to typeset some of the above examples properly. The extra thin space at the right of the ellipsis is correct in certain situations (e.g., when a punctuation character follows). If the ellipsis is followed by space, however, it looks distinctly odd and is best canceled as shown in the example below (though removing the space in the second instance brings the exclamation mark a bit too close).

Compare the following:
Du E... du! Scher dich zum ... !
Du E... du! Scher dich zum ...!

3-1-2

```
\newcommand\lips{\dots\unkern}
Compare the following:\\
  Du E\dots\ du! Scher dich zum \dots!\\
  Du E\lips\ du! Scher dich zum \lips!
```

This problem is addressed in the package ellipsis written by Peter Heslin, which redefines the `\dots` command to look at the following character to decide whether to add a final separation. An extra space is added if the following character is listed in the command `\ellipsispunctuation`, which defaults to "`,.:;!?`". When using some of the language support packages that make certain characters active, this list may have to be redeclared afterwards to enable the package to still recognize the characters.

The spacing between the periods and the one possibly added after the ellipsis can be controlled through the command `\ellipsisgap`. To allow for automatic adjustments depending on the font size use a font-dependent unit like `em` or a fraction of a `\fontdimen` (see page 428).

Compare the following:
Du E... du! Scher dich zum ...!
Du E. . . du! Scher dich zum . . . !
Du E. . . du! Scher dich zum . . . !

```
\usepackage{ellipsis}
Compare the following:\\
  Du E\dots\ du! Scher dich zum \dots!\\
\renewcommand\ellipsisgap{1.5\fontdimen3\font}
  Du E\dots\ du! Scher dich zum \dots!\\
\renewcommand\ellipsisgap{0.3em}
  Du E\dots\ du! Scher dich zum \dots!
```

3-1-3

For the special case when you need an ellipsis in the middle of a word (or for other reasons want a small space at either side), the package offers the command `\midwordellipsis`. If the package is loaded with the option `mla` (Modern Language Association style), the ellipsis is automatically bracketed without any extra space after the final period.

If one follows *The Chicago Manual of Style* [38], then an ellipsis is set with full word spaces between the dots. For this, one can deploy the lips package[1] by Matt Swift. It implements the command `\lips`, which follows the recommendations in this reference book. For example, an ellipsis denoting an omission at the end of a sentence should, according to [38, §10.48–63], consist of four dots with the *first* dot being the sentence period.[2] The `\lips` command implements this by interpreting "`\lips.`" like "`.\lips`", as can be seen in the next example.

Elsewhere . . . the dots are normally set with full word spaces between them. . . . An example would be this paragraph.

```
\usepackage{moredefs,lips}
Elsewhere \lips the dots are normally set with
full word spaces between them \lips. An example
would be this paragraph.
```

3-1-4

The `\lips` command looks for punctuation characters following it and ensures that in case of `,:;?!)’]/` the ellipsis and the punctuation are not separated by a line break. In other cases (e.g., an opening parenthesis), a line break would be possible. The above list is stored in `\LPNobreakList` and can be adjusted if

[1] lips is actually part of a larger suite of packages. If used on a stand-alone basis, you also have to load the package moredefs by the same author.

[2] Not that the authors of this book can see any logic in this.

necessary. To force an unbreakable space following `\lips`, follow the command with a tie (`~`).

When applying the `mla` option the ellipsis generated will be automatically bracketed and a period after the `\lips` command will not be moved to the front. If necessary, `\olips` will produce the original unbracketed version.

Elsewhere . . . the dots are normally set with full word spaces between them [. . .]. An example would be this paragraph. 3-1-5

```
\usepackage{moredefs}\usepackage[mla]{lips}
Elsewhere \olips the dots are normally set with
full word spaces between them \lips. An example
would be this paragraph.
```

3.1.3 amsmath—Nonbreaking dashes

The amsmath package, extensively discussed in Chapter 8, also offers one command for use within paragraphs. The command `\nobreakdash` suppresses any possibility of a line break after the following hyphen or dash. A very common use of `\nobreakdash` is to prevent undesirable line breaks in usages such as “p-adic” but here is another example: if you code “Pages 3-9” as `Pages 3\nobreakdash--9` then a line break will never occur between the dash and the 9.

This command must be used *immediately* before a hyphen or dash (`-`, `--`, or `---`). The following example shows how to prohibit a line break after the hyphen but allow normal hyphenation in the following word (it suffices to add a zero-width space after the hyphen). For frequent use, it’s advisable to make abbreviations, such as `\p`. As a result “dimension” is broken across the line, while a break after “p-” is prevented (resulting in a overfull box in the example) and “3-9” is moved to the next line.

The generalization to the n-dimensional case (using the standard p-adic topology) can be found on Pages 3–9 of Volume IV. 3-1-6

```
\usepackage{amsmath}
\newcommand\p{$p$\nobreakdash}%      "\p-adic"
\newcommand\Ndash{\nobreakdash--}% "3\Ndash 9"
\newcommand\n[1]{$n$\nobreakdash-\hspace{0pt}}
                                 % "\n-dimensional"
\noindent  The generalization to the \n-dimensional
case (using the standard \p-adic topology) can be found
on Pages 3\Ndash 9 of Volume IV.
```

3.1.4 relsize—Relative changes to the font size

Standard LaTeX offers 10 predefined commands that change the overall font size (see Table 7.1 on page 342). The selected sizes depend on the document class but are otherwise absolute in value. That is, `\small` will always select the same size within a document regardless of surrounding conditions.

However, in many situations it is desirable to change the font size relative to the current size. This can be achieved with the relsize package, originally developed by Bernie Cosell and later updated and extended for LaTeX 2ε by Donald Arseneau and Matt Swift.

The package provides the declarative command `\relsize`, which takes a number as its argument denoting the number of steps by which to change the size. For example, if the current size is `\large` then `\relsize{-2}` would change to `\small`. If the requested number of steps is not available then the smallest (i.e., `\tiny`) or largest (i.e., `\Huge`) size command is selected. This means that undoing a relative size change by negating the argument of `\relsize` is not guaranteed to bring you back to the original size—it is better to delimit such changes by a brace group and let LaTeX undo the modification.

The package further defines `\smaller` and `\larger`, which are simply abbreviations for `\relsize` with the arguments `-1` and `1`, respectively. Convenient variants are `\textsmaller` and `\textlarger`, whose argument is the text to reduce or enlarge in size. These four commands take as an optional argument the number of steps to change if something different from 1 (the default) is needed.

Some large text with a few small words inside.

Small Caps (faked)
Small Caps (real; compare the running length and stem thickness to previous line).

```
\usepackage{relsize}
\Large Some large text with  a few
    {\relsize{-2} small words} inside.
\par\medskip
\normalsize\noindent
S\textsmaller[2]{MALL} C\textsmaller[2]{APS} (faked)\\
\textsc{Small Caps} (real; compare the running length
     and stem thickness to previous line).
```

3-1-7

In fact, the above description for `\relsize` is not absolutely accurate: it tries to increase or decrease the size by 20% for each step and selects the LaTeX font size command that is closest to the resulting target size. It then compares the selected size and target size. If they differ by more than the current value of `\RSpercentTolerance` (interpreted as a percentage), the package calls `\fontsize` with the target size as one of the arguments. If this happens it is up to LaTeX's font selection scheme to find a font matching this request as closely as possible. By default, `\RSpercentTolerance` is an empty macro, which is interpreted as 30 (percent) when the current font shape group is composed of only discrete sizes (see Section 7.10.3), and as 5 when the font shape definition covers ranges of sizes.

Using a fixed factor of `1.2` for every step may be too limiting in certain cases. For this reason the package additionally offers the more general declarative command `\relscale{`*factor*`}` and its variant `\textscale{`*factor*`}{`*text*`}`, to select the size based on the given *factor*, such as `1.3` (enlarge by 30%).

There are also two commands, `\mathsmaller` and `\mathlarger`, for use in math mode. LaTeX recognizes only four different math sizes, of which two (`\displaystyle` and `\textstyle`) are nearly identical for most symbols, so the application domain of these commands is somewhat limited. With exscale additionally loaded the situation is slightly improved: the `\mathlarger` command, when used in `\displaystyle`, will then internally switch to a larger text font size and afterwards select the `\displaystyle` corresponding to that size.

3-1-8

$$\sum \neq \sum$$

and $\frac{1}{2} \neq \frac{1}{2}$ but $N = N$

```
\usepackage{exscale,relsize}
\[ \sum \neq \mathlarger{\sum} \]
and $\frac{1}{2} \neq \frac{\mathlarger 1}
{2}$ but $N = \mathlarger {N}$
```

These commands will attempt to correctly attach superscripts and subscripts to large operators. For example,

3-1-9

$$\textstyle\sum_{i=1}^n \neq \displaystyle\sum_{i=1}^n \neq \sum_{i=1}^n \qquad \textstyle\int_0^\infty \neq \displaystyle\int_0^\infty \neq \int_0^\infty$$

```
\usepackage{exscale,relsize}
\[ \mathsmaller\sum_{i=1}^n \neq
 \sum_{i=1}^n \neq \mathlarger\sum_{i=1}^n
\qquad  \mathsmaller\int_0^\infty \neq
 \int_0^\infty \neq \mathlarger\int_0^\infty
\]
```

Be aware that the use of these commands inside formulas will hide the true nature of the math atoms inside the argument, so that the spacing in the formula, without further help, might be wrong. As shown in following example, you may have to explicitly use `\mathrel`, `\mathbin`, or `\mathop` to get the correct spacing.

3-1-10

$$a \times b \neq a{\times}b \neq a \times b$$

```
\usepackage{exscale,relsize}
\[ a \times b \neq  a \mathlarger{\times} b \neq
   a \mathbin{\mathlarger\times} b \]
```

Due to these oddities, the `\mathlarger` and `\mathsmaller` commands should not be trusted blindly, and they will not be useful in every instance.

3.1.5 textcase—Change case of text intelligently

The standard LaTeX commands `\MakeUppercase` and `\MakeLowercase` change the characters in their arguments to uppercase or lowercase, respectively, thereby expanding macros as needed. For example,

```
    \MakeUppercase{On \today}
```

will result in “ON 28TH OF JULY 2003”. Sometimes this will change more characters than desirable. For example, if the text contains a math formula, then uppercasing this formula is normally a bad idea because it changes its meaning. Similarly, arguments to the commands `\label`, `\ref`, and `\cite` represent semantic information, which, if modified, will result in incorrect or missing references, because LaTeX will look for the wrong labels.

```
\MakeTextUppercase{text}     \MakeTextLowercase{text}
```

The package textcase by David Carlisle overcomes these defects by providing two alternative commands, `\MakeTextUppercase` and `\MakeTextLowercase`, which recognize math formulas and cross-referencing commands and leave them alone.

1 Textcase example

TEXT IN SECTION 1, ABOUT $a = b$ AND $\alpha \neq a$

```
\usepackage{textcase}
\section{Textcase example}\label{exa}
\MakeTextUppercase{Text in section~\ref{exa},
  about $a=b$ and \(\alpha \neq a \) }
```

3-1-11

Sometimes portions of text should be left unchanged for one reason or another. With `\NoCaseChange` the package provides a generic way to mark such parts. For instance:

SOME TEXT Some More TEXT

```
\usepackage{textcase}
\MakeTextUppercase{Some text
     \NoCaseChange{Some More} text}
```

3-1-12

If necessary, this method can be used to hide syntactic information, such as

```
\NoCaseChange{\begin{tabular}{ll}} ... \NoCaseChange{\end{tabular}}
```

thereby preventing `tabular` and `ll` from incorrectly being uppercased.

All this works only as long as the material is on the top level. Anything that is inside a group of braces (other than the argument braces to `\label`, `\ref`, `\cite`, or `\NoCaseChange`) will be uppercased or lowercased regardless of its nature.

BOTH OF THESE WILL **FAIL** $A + B = C$ *UNFORTUNATELY*

```
\usepackage{textcase}
\MakeTextUppercase{Both of these will
  \textbf{fail $a+b=c$}
  \emph{\NoCaseChange{unfortunately}}}
```

3-1-13

In the above case you could avoid this pitfall by taking the formula out of the argument to `\textbf` and moving `\emph` inside the argument to `\NoCaseChange`. In other situations this kind of correction might be impossible. In such a case the (somewhat cumbersome) solution is to hide the problem part inside a private macro and protect it from expansion during the case change; this method works for the standard LaTeX commands as well, as shown in the next example.

BUT THIS WILL **WORK** $a + b = c$ ALWAYS

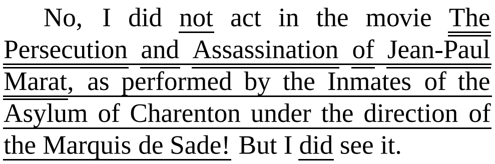
```
\newcommand\mymath{$a+b=c$}
\MakeUppercase{But this will
  \textbf{work \protect\mymath} always}
```

3-1-14

Some classes and packages employ `\MakeUppercase` internally—for example, in running headings. If you wish to use `\MakeTextUppercase` instead, you should load the textcase package with the option overload. This option will replace the standard LaTeX commands with the variants defined by the package.

3.1.6 ulem—Emphasize via underline

LaTeX encourages the use of the \emph command and the \em declaration for marking emphasis, rather than explicit font-changing declarations, such as \bfseries and \itshape. The ulem package (by Donald Arseneau) redefines the command \emph to use underlining, rather than italics. It is possible to have line breaks and even primitive hyphenation in the underlined text. Every word is typeset in an underlined box, so automatic hyphenation is normally disabled, but explicit discretionary hyphens (\-) can still be used. The underlines continue between words and stretch just like ordinary spaces do. As spaces delimit words, some difficulty may arise with syntactical spaces (e.g., "2.3 pt"). Some effort is made to handle such spaces. If problems occur you might try enclosing the offending command in braces, since everything inside braces is put inside an \mbox. Thus, braces suppress stretching and line breaks in the text they enclose. Note that nested emphasis constructs are not always treated correctly by this package (see the gymnastics performed below to get the interword spaces correct in which each nested word is put separately inside an \emph expression).

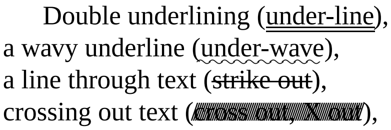


No, I did not act in the movie The Persecution and Assassination of Jean-Paul Marat, as performed by the Inmates of the Asylum of Charenton under the direction of the Marquis de Sade! But I did see it.

3-1-15

```
\usepackage{ulem}
No, I did \emph{not} act in the movie
\emph{\emph{The} \emph{Persecution} \emph{and}
\emph{Assassination} \emph{of} \emph{Jean-Paul}
\emph{Marat}, as performed  by the Inmates of
the Asylum of Charenton under the direc\-tion of
the Marquis de~Sade!} But I \emph{did} see it.
```

Alternatively, underlining can be explicitly requested using the \uuline command. In addition, a number of variants are available that are common in editorial markup. These are shown in the next example.



Double underlining (under-line),
a wavy underline (under-wave),
a line through text (~~strike out~~),
crossing out text (~~cross out, X out~~),

3-1-16

```
\usepackage{ulem}
Double underlining  (\uuline{under-line}),\\
a wavy underline    (\uwave{under-wave}), \\
a line through text (\sout{strike out}),  \\
crossing out text (\xout{cross out, X out}),
```

The redefinition of \emph can be turned off and on by using \normalem and \ULforem. Alternatively, the package can be loaded with the option normalem to suppress this redefinition. Another package option is UWforbf, which replaces \textbf and \bfseries by \uwave whenever possible.

The position of the line produced by \uline can be set explicitly by specifying a value for the length \ULdepth. The default value is font-dependent, denoted by the otherwise senile value `\maxdimen`. Similarly, the thickness of the line can be controlled via `\ULthickness`, which, for some historical reason, needs to be redefined using `\renewcommand`.

3.1.7 soul—Letterspacing or stealing sheep

Frederic Goudy supposedly said, “Anyone who would letterspace black letter would steal sheep”. Whether true or a myth, the topic of letterspacing clearly provokes heated discussions among typographers and is considered bad practice in most situations because it changes the “grey” level of the text and thus disturbs the flow of reading. Nevertheless, there are legitimate reasons for undertaking letterspacing. For example, display type often needs a looser setting and in most fonts uppercased text is improved this way. You may also find letterspacing being used to indicate emphasis, although this exhibits the grey-level problem.

TEX is ill equipped when it comes to supporting letterspacing. In theory, the best solution is to use specially designed fonts rather than trying to solve the problem with a macro package. But as this requires the availability of such fonts, it is not an option for most users. Thus, in practice, the use of a macro-based solution is usually easier to work with, even though it means dealing with a number of restrictions. Some information about the font approach can be found in the documentation for the fontinst package [74, 75].

The soul package written by Melchior Franz provides facilities for letterspacing and underlining, but maintains TEX’s ability to automatically hyphenate words, a feature not available in ulem. The package works by parsing the text to be letterspaced or underlined, token by token, which results in a number of peculiarities and restrictions. Thus, users who only wish to underline a few words and do not need automatic hyphenation are probably better off with ulem, which is far less picky about its input.

```
\caps{text}     \hl{text}     \so{text}     \st{text}     \ul{text}
```

The use of the five main user commands of soul are shown in the next example. In cases where TEX’s hyphenation algorithm fails to find the appropriate hyphenation points, you can guide it as usual with the `\-` command. If the color package is loaded, `\hl` will work like a text marker, coloring the background using yellow as the default color; otherwise, it will behave like `\ul` and underline its argument.

With the soul package you can letter-space words and phrases. Capitals are LETTERSPACED with a different command. Interfaces for underlining, ~~strikeouts~~, and highlighting are also provided.

```
\usepackage{soul,color}
With the \texttt{soul} package you can
\so{letter\-space words and phrases}. Capitals
are \caps{LETTERSPACED} with a different
command. Interfaces for \ul{underlining},
\st{strikeouts}, and \hl{highlighting} are
also provided.
```

3-1-17

Normally, the soul package interprets one token after another in the argument of \so, \st, and so on. However, in case of characters that are represented by more than one token (e.g., accented characters) this might fail with some low-level TEX error messages. Fortunately, the package already knows about all common accent commands, so these are handled correctly. For others, such as those provided by the textcomp package, you can announce them to soul with the help of a \soulaccent declaration. The alternative is to surround the tokens by braces.

3-1-18 ~~ä ù Õ X̀ Y̆~~

```
\usepackage{soul} \usepackage{textcomp}
\soulaccent{\capitalgrave}
\Huge \st{\"a \`u \~O \capitalgrave X {\capitalbreve Y}}
```

The soul package already knows that quotation characters, en dash, and em dash consist of several tokens and handles them correctly. In case of other syntactical ligatures, such as the Spanish exclamation mark, you have to help it along with a brace group.

3-1-19 “So there,” he said. ¡HOLA—MY **FRIEND**!

```
\usepackage{soul}
\so{``So there,''} he said. \caps{{!`}Hola---my \textbf{friend}!}
```

The soul package also knows about math formulas as long as they are surrounded by $ signs (the form \(...\) is not supported) and it knows about all standard font-changing commands, such as \textbf. If you have defined your own font-switching command or use a package that provides additional font commands, you have to register them with soul using \soulregister. This declaration expects the font command to be registered as its first argument and the number of arguments (i.e., 0 or 1) for that command to appear as its second argument. Within the soul commands none of the font commands inserts any (necessary) italic correction. If needed, one has to provide it manually using \/.

3-1-20 Here we see **soul** in *action*: $x \neq y$ OK?

```
\newcommand\textsfbf[1]{\textsf{\bfseries#1}}
\usepackage{soul} \soulregister{\textsfbf}{1}
\so{Here we see \textsfbf{soul} in \emph{action}: $x\neq y$ OK?}
```

If you look carefully, you will see that the font commands suppress letterspacing directly preceding and following them, such as between “action” and the colon. This can be corrected by adding \>, which forces a space.

3-1-21 bl**oo**dy viz. bl **oo** dy

```
\usepackage{soul}
\so{bl\textbf{oo}dy  viz. bl\>\textbf{oo}\>dy}
```

Text inside a brace group is regarded as a single object during parsing and is therefore not spaced out. This is handy if certain ligatures are to be kept intact inside spaced-out text. However, this method works only if the text inside the brace group contains no hyphenation points. If it does, you will receive the package error message “Reconstruction failed”. To hide such hyphenation points you need to put the text inside an \mbox, as shown in the second text line of the next example (TEX would hyphenate this as “Es-cher”—that is, between the “sch” that we try to keep together). You can also use \soulomit to achieve this effect, but then your text will work only when the soul package is loaded.

Schutzvorrichtung
Gödel, Escher, Bach
Temporarily disabling the scanner

```
\usepackage{soul,yfonts} \usepackage[latin1]{inputenc}
\textfrak{\so{S{ch}u{tz}vorri{ch}tung}} \par
\so{Gödel, E\mbox{sch}er, Bach}         \par
\ul{Temporarily dis\soulomit{abl}ing the scanner}
```

3-1-22

One of the most important restrictions of the above commands is that they cannot be nested; any attempt to nest soul commands will result in low-level TEX errors. If you really need nesting you will have to place the inner material in a box, which means you lose the possibility to break the material at a line ending.

This is hell for all of us!

```
\usepackage{soul}  \newsavebox\soulbox
\sbox\soulbox{\so{ is hell }}
\ul{This\mbox{\usebox{\soulbox}}for all of us!}
```

3-1-23

A few other commands are special within the argument of \so and friends. Spacing out at certain points can be canceled using \< or forced with \> as we saw above. As usual with LATEX a ~ will produce an unbreakable space. The \\ command is supported, though only in its basic form—no star, no optional argument. You can also use \linebreak to break a line at a certain point, but again the optional argument is not supported. Other LATEX commands are likely to break the package—some experimentation will tell you what is safe and what produces havoc. The next example shows applications of these odds and ends.

“So there” he said. Let’s
produce a spaced out line,
OK?

```
\usepackage{soul}
\so{‘‘\<So there\<’’ he said.  Let’s\\
 produce a spaced out line\>,\linebreak OK?}
```

3-1-24

\sodef{*cmd*}{*font*}{*inter-letter space*}{*word space*}{*outer space*}

The \sodef declaration allows you to define your own letterspacing commands. It can also be used to overwrite the defaults for \so.

The letterspacing algorithm works by putting a certain *inter-letter space* between characters of a word, a certain *word space* between words, and a certain *outer space* at the beginning and end of the letterspaced text section. The latter space is added only if it is appropriate at that point. The default values for these spaces are adjusted for typesetting texts in Fraktur fonts but with the help of the \sodef declaration it is easy to adjust them for your own needs. The *font* argument allows you to specify font attributes; in most cases it will be empty. Rather than using explicit dimensions in the other arguments it is advisable to resort to em values, thereby making your definition depend on the current font and its size.

3-1-25 Here we **emphasize words** a lot.

```
\usepackage{soul}
\sodef\sobf{\bfseries}{.3em}{1em plus .1em}
                                {1.3em plus.1em minus.2em}
Here we \sobf{emphasize words} a lot.
```

While \so or any new command defined via \sodef simply retrieves and executes its stored definition, the \caps command works somewhat differently. It examines the current font and tries to find it (or a close match) in an internal database. It then uses the letterspacing values stored there. You can extend this database using the \capsdef declaration by providing values for individual fonts or groups of fonts. In this way you can fine-tune the letterspacing—for example, for text in headings. It is even possible to keep several such databases and change them on the fly within a document.

\capsdef{*match spec*}{*font*}{*inter-letter space*}{*word space*}{*outer space*}

Apart from the first argument, which is totally different, the other arguments to \capsdef are identical to those of \sodef. The first argument, *match spec*, defines the font (or fonts) to which the current declaration applies.

Its syntax is *encoding*, *family*, *series*, *shape*, and *size* separated by slashes using the naming conventions of NFSS. Empty values match anything, so //// matches any font, /ptm///10 matches all Times fonts in 10 points, and OT1/cmr/m/n/ matches Computer Modern (cmr) medium series (m) normal shape (n) encoded in OT1 in any size. It is also possible to specify size ranges. For example, 5-14 means 5 pt ≤ *size* < 14 pt and 14- matches all sizes equal or greater 14 pt. Refer to the tables in Chapter 7 for details on the NFSS font naming conventions.

As with \sodef, in most declarations the *font* argument will be empty. On some occasions it may make sense to use \scshape in this place, such as to change the font shape to small caps before applying letterspacing.

Because \caps uses the first matching entry in its database, the order of \capsdef declarations is important. Later declarations are examined first so that it is possible to overwrite or extend existing declarations.

A SAMPLE HEADING

The \capsdef declaration applies here, because the heading definition specifies sans serif and our examples are typeset with Times and Helvetica (phv). 3-1-26

```
\usepackage{titlesec,soul}
\newcommand\allcaps[1]{\MakeUppercase{\caps{#1}}}
\titleformat{\section}[block]{\centering\sffamily}
                              {\thesection.}{.5em}{\allcaps}
\titlespacing*{\section}{0pt}{8pt}{3pt}
\capsdef{/phv///}{\scshape}{.17em}{.55em}{.4em}
\section*{A Sample Heading}
The \verb=\capsdef= declaration applies here, because the
heading definition specifies sans serif and our examples
are typeset with Times and Helvetica (\texttt{phv}).
```

The previous example also contained an interesting combination of `\caps` and `\MakeUppercase`: the command `\allcaps` changes its argument to uppercase and then uses `\caps` to letterspace the result.

```
\capssave{name}    \capsselect{name}    \capsreset
```

Customized letterspacing for different occasions

With `\capsreset` the database is restored to its initial state containing only a generic default. You can then add new entries using `\capsdef`. The current state of the `\caps` database can be stored away under a *name* by using `\capssave`. You can later retrieve this state by recalling it with `\capsselect`. If you use the `capsdefault` option when loading the package, then all uses of `\caps` that have no matching declaration are flagged by underlining the text.

A SAMPLE HEADING

Notice the different letterspacing in the heading and RUNNING TEXT. For Times we have no definition above so that the <u>DEFAULT</u> will match.

```
\usepackage{titlesec} \usepackage[capsdefault]{soul}
\capsdef{/phv///}{\scshape}{.17em}{.55em}{.4em}
\capssave{display}    \capsreset
\capsdef{/phv///}{\scshape}{.04em}{.35em}{.35em}
\titlespacing*{\section}{0pt}{8pt}{3pt}
\titleformat{\section}[block]{\centering\sffamily}
            {\thesection.}{.5em}{\capsselect{display}\caps}
\section*{A Sample Heading}
Notice the different letterspacing in the heading and
\textsf{\caps{Running Text}}. For Times we have no
definition above so that the \caps{default} will match.
```

3-1-27

Customizing underlining

The position and the width of the line produced by the `\ul` command can be customized using either `\setul` or `\setuldepth`. The command `\setul` takes two dimensions as arguments: the position of the line in relation to the baseline and the width of the line. Alternatively, `\setuldepth` can be used to specify that the line should be positioned below the text provided as an argument. Finally, `\resetul` will restore the default package settings.

<u>Here we test</u>
~~a number of~~
<u>different settings.</u>
<u>And back to normal!</u>

```
\usepackage{soul}
\setul{0pt}{.4pt}    \ul{Here we test}         \par
\setul{-.6ex}{.3ex}  \ul{a number of}          \par
\setuldepth{g}       \ul{different settings.}  \par
\resetul             \ul{And back to normal!}
```

3-1-28

Both `\ul` and `\st` use a black rule by default. If you additionally load the color package, you can use colored rules instead and, if desired, modify the highlighting color:

Rules can be in ~~black~~ <u>blue</u>.

```
\usepackage{soul,color}
\sethlcolor{green} \setulcolor{blue} \setstcolor{red}
Rules \hl{can} be in \st{black} \ul{blue}.
```

3-1-29

3.1.8 url—Typesetting URLs, path names, and the like

E-mail addresses, URLs, path or directory names, and similar objects usually require some attention to detail when typeset. For one thing, they often contain characters with special significance to LaTeX, such as `~`, `#`, `&`, `{`, or `}`. In addition, breaking them across lines should be avoided or at least done with special care. For example, it is usually not wise to break at a hyphen, because then it is not clear whether the hyphen was inserted because of the break (as it would be the case with normal words) or was already present. Similar reasons make breaks at a space undesirable. To help with these issues, Donald Arseneau wrote the url package, which attempts to solve most of these problems.

```
\url{text}   \url!text!          \path{text}   \path=text=
```

The base command provided by the package is `\url`, which is offered in two syntax variants: the *text* argument either can be surrounded by braces (in which case the *text* must not contain unbalanced braces) or, like `\verb`, can be delimited by using an arbitrary character on both sides that is not used inside *text*. (The syntax box above uses `!` and `=` but these are really only examples.) In that second form one can have unbalanced braces in the argument.

The `\path` command is the same except that it always uses typewriter fonts (`\ttfamily`), while `\url` can be customized as we will see below. The argument to both commands is typeset pretty much verbatim. For example, `\url{~}` produces a tilde. Spaces are ignored by default, as can be seen in the following example.

The LaTeX project web pages are at `http://www.latex-project.org` and my home directory is `~frank` (sometimes).

3-1-30

```
\usepackage{url}
The \LaTeX{} project web pages are at
\url{http://www . latex-project . org} and my
home directory is \path+~frank+ (sometimes).
```

Line breaks can happen at certain symbols (by default, not between letters or hyphens) and in no case can the commands add a hyphen at the break point. Whenever the *text* contains either of the symbols `%` or `#`, or ends with `\`, it cannot be used in the argument to another command without producing errors (just like the `\verb` command). Another case that does not work properly inside the argument of another command is the use of two `^` characters in succession. However, the situation is worse in that case because one might not even get an error but simply incorrect output[1] as the next example shows.

`^frank` and `^frank` (OK)
`^^frank` but &rank (bad)

3-1-31

```
\usepackage{url}
\url{^frank}  and \mbox{\url{^frank}}  (OK)\par
\url{^^frank} but \mbox{\url{^^frank}} (bad)
```

[1] It depends on the letter that is following. An uppercase `F` instead of the lowercase `f` would produce an error.

Even if the *text* does not contain any critical symbols, it is always forbidden to use such a command inside a moving argument—for instance, the argument of a `\section`. If used there, you will get the error message

```
! Undefined control sequence.
\Url Error ->\url used in a moving argument.
```

followed by many strange errors. Even the use of `\protect` will not help in that case. So what can be done if one needs to cite a path name or a URL in such a place? If you are prepared to be careful and only use "safe" characters inside *text*, then you can enable the commands for use in moving arguments by specifying the option `allowmove` when loading the package. But this does not help if you actually need a character like "#". In that case the solution is to record the information first using `\urldef` and then reuse it later.

`\urldef{`*cmd*`}{`*url-cmd*`}{`*text*`}`   `\urldef{`*cmd*`}{`*url-cmd*`}=`*text*`=`

The declaration `\urldef` defines a new command *cmd* to contain the *url-cmd* (which might be `\url`, `\path`, or a newly defined command—see below) and the *text* in a way such that they can be used in any place, including a moving argument. The *url-cmd* is not executed at this point, which means that style changes can still affect the typesetting (see Example 3-1-33 on the facing page). Technically, what happens is that the `\catcode`s of characters in *text* are frozen during the declaration, so that they cannot be misinterpreted in places like arguments.

1   `^^frank~#$\` works?

It does—in contrast to the earlier example.

```
\usepackage{url}
\urldef\test\path{^^frank~#$\}
\section{\test{} works?}
It does---in contrast to the earlier example.
```

3-1-32

`\urlstyle{`*style*`}`

We have already mentioned style changes. For this task the url package offers the `\urlstyle` command, which takes one mandatory argument: a named *style*. Predefined styles are `rm`, `sf`, `tt`, and `same`. The first three select the font family of that name, while the `same` style uses the current font and changes only the line breaking.

The `\url` command uses whatever style is currently in force (the default is `tt`, i.e., typewriter), while `\path` internally always switches to the `tt` style. In the following example we typeset a URL saved in `\lproject` several times using different styles. The particular example may look slightly horrifying, but imagine how it would have looked if the URL had not been allowed to split at all in this narrow measure.

Zapf Chancery! `http://www.latex-project.org` *(default setup)* http://www.latex-project.org *(CM Roman)* http://www.latex-project.org *(CM Sans Serif)* `http://www.latex-project.org` *(CM Typewriter)* *http://www.latex-project.org (Zapf Chancery)*

3-1-33

```
\usepackage[hyphens]{url}
\urldef\lproject\url{http://www.latex-project.org}
\fontfamily{pzc}\selectfont Zapf Chancery!
             \lproject\   (default setup) \quad
\urlstyle{rm}\lproject\   (CM Roman)      \quad
\urlstyle{sf}\lproject\   (CM Sans Serif) \quad
\urlstyle{tt}\lproject\   (CM Typewriter) \quad
\urlstyle{same}\lproject\ (Zapf Chancery)
```

If you studied the previous example closely you will have noticed that the option `hyphens` was used. This option allows breaking at explicit hyphens, something normally disabled for `\url`-like commands. Without this option breaks would have been allowed only at the periods, after the colon, or after “`//`”.

Spaces in the argument

As mentioned earlier spaces inside *text* are ignored by default. If this is not desired one can use the option `obeyspaces`. However, this option may introduce spurious spaces if the `\url` command is used inside the argument of another command and *text* contains any “`\`” character. In that case `\urldef` solves the problem. Line breaks at spaces are not allowed unless you also use the option `spaces`.

The package automatically detects which font encoding is currently in use. In case of `T1` encoded fonts it will make use of the additional glyphs available in this encoding, which improves the overall result.

Appending material at left or right

The package offers two hooks, `\UrlLeft` and `\UrlRight`, that by default do nothing but can be redefined to typeset material at the left or right of *text*. The material is typeset in the same fashion as the *text*. For example, spaces are ignored unless one uses `\␣` or specifies `obeyspaces` as an option. If the commands are redefined at the top level, they act on every `\url`-like command. See Example 3-1-34 on the next page for a possibility to restrict their scope.

`\DeclareUrlCommand{`*cmd*`}{`*style-information*`}`

Defining URL-like commands

It is sometimes helpful to define your own commands that work similarly to `\url` or `\path` but use their own fonts, and so on. The command `\DeclareUrlCommand` can be used to define a new `\url`-like command or to modify an existing one. It takes two arguments: the command to define or change and the *style-information* (e.g., `\urlstyle`).

In the next example, we define `\email` to typeset e-mail addresses in `rm` style, prepending the string “e-mail: ” via `\UrlLeft`. The example clearly shows that the scope for this redefinition is limited to the `\email` command. If you look closely, you can see that a space inside `\UrlLeft` (as in the top-level definition) has no effect, while `\␣` produces the desired result.

<url:http://www.latex-project.org>
e-mail: frank.mittelbach@latex-project.org
<url:$HOME/figures> oops, wrong!

```
\usepackage{url}
\renewcommand\UrlLeft{<url: }
\renewcommand\UrlRight{>}
\DeclareUrlCommand\email{\urlstyle{rm}%
   \renewcommand\UrlLeft{e-mail:\ }%
   \renewcommand\UrlRight{}}
\url{http://www.latex-project.org}         \par
\email{frank.mittelbach@latex-project.org} \par
\path{$HOME/figures} oops, wrong!
```

3-1-34

The url package offers a number of other hooks that influence line breaking, among them `\UrlBreaks`, `\UrlBigBreaks`, and `\UrlNoBreaks`. These hooks can be redefined in the *style-information* argument of `\DeclareUrlCommand` to set up new or special conventions. For details consult the package documentation, which can be found at the end of the file `url.sty`.

3.1.9 euro—Converting and typesetting currencies

To ease the calculations needed to convert between national units and the euro, Melchior Franz developed the package euro. In fact, the package converts arbitrary currencies using the euro as the base unit. The calculations are done with high precision using the fp package written by Michael Mehlich. The formatting is highly customizable on a per-currency basis, so that this package can be used for all kind of applications involving currencies whether or not conversions are needed.

\EURO{*from-currency*}[*to-currency*]{*amount*}

The main command `\EURO` converts an *amount* in *from-currency* into *to-currency* or, if this optional argument is missing, into euros. The arguments *from-currency* and *to-currency* are denoted in ISO currency codes, as listed in Table 3.1 on the facing page. When inputting the *amount* a dot must separate the integer value from any fractional part, even if the formatted number uses a different convention.

With the default settings the *amount* is displayed in the *from-currency* with the converted value in the *to-currency* shown in parentheses.

7 DM (23,48 FRF)   23,48 FRF (7 DM)
10 Euro (19,56 DM)   20 DM (10,23 Euro)

```
\usepackage{euro}
\EURO{DEM}[FRF]{7}\quad \EURO{FRF}[DEM]{23.48}
\\
\EURO{EUR}[DEM]{10.00}\quad \EURO{DEM}{20}
```

3-1-35

EUR	Europe	GRD	Greece
ATS	Austria	IEP	Ireland
BEF	Belgium	ITL	Italy
DEM	Germany	LUF	Luxembourg
ESP	Spain	NLG	The Netherlands
FIM	Finland	PTE	Portugal
FRF	France		

Table 3.1: ISO currency codes of the *euro* and the 12 *euro-zone* countries

The package options

The package offers a number of options to influence the general style of the output (unless overwritten by the more detailed formatting declarations discussed below). With `eco` the ISO codes precede the value and no customized symbols are used; with `dots` a period is inserted between every three-digit group (the default is to use a small space).

By default, integer amounts are printed as such, without adding a decimal separator and a (zero) fractional part. If the `table` option is specified this behavior is globally changed and either a — (option `emdash`, also the default), a – (option `endash`), or the right number of zeros (option `zeros`) is used.

DEM 7,– (FRF 23,48) FRF 23,48 (DEM 7,–)
3-1-36 EUR 10,– (DEM 19,56) DEM 20,– (EUR 10,23)

```
\usepackage[eco,table,endash]{euro}
\EURO{DEM}[FRF]{7}\quad \EURO{FRF}[DEM]{23.48}
\\ \EURO{EUR}[DEM]{10.00}\quad \EURO{DEM}{20}
```

The more detailed output customizations, which we discuss below, can be placed anywhere in the document. It is, however, advisable to keep them together in the preamble, or even to put them into the file `euro.cfg`, which is consulted upon loading the package.

The monetary symbols typeset can be adjusted with a `\EUROSYM` declaration; as defaults the package uses the ISO codes for most currencies. The example below changes the presentation for lira and euro using the currency symbols from the textcomp package. It also uses `dots` to help with huge lira amounts.

3-1-37 10.000 ₤ (5,16 €) 1.000 DM (989.999 ₤)

```
\usepackage{textcomp}\usepackage[dots]{euro}
\EUROSYM{ITL}{\textlira}\EUROSYM{EUR}{\texteuro}
\EURO{ITL}{10000}\quad \EURO{DEM}[ITL]{1000}
```

The package is well prepared for new countries to join the euro-zone. In fact, it is well prepared to deal with conversions from and to any currency as long as the conversion rate to the euro is known. To add a new currency use the `\EUROADD` declaration, which takes three arguments: the ISO currency code, the symbol or text to display for the currency, and the conversion rate to the euro. The next example makes the British pound available. Note the abbreviation \GBP, which makes the input a bit easier.

14,90 £ (23,29 €)
10 £ (102,54 FRF)
10 € (6,40 £)

```
\usepackage{eurosans,euro}
\EUROADD{GBP}{\textsterling}{0.6397} % 2002/12/21
\newcommand*\GBP{\EURO{GBP}} \EUROSYM{EUR}{\euro}
\noindent \GBP{14.9}\\ \GBP[FRF]{10}\\ \EURO{EUR}[GBP]{10}
```

3-1-38

The conversion rates for the national currencies of the euro-zone countries are fixed (and predefined by the package). With other currencies the rates may change hourly, so you have to be prepared for frequent updates.

The package allows you to tailor the presentation via \EUROFORMAT declarations, either to provide new defaults or to adjust the typesetting of individual currencies. The first argument specifies which part of the formatting should be adjusted, and the second argument describes the formatting.

The main format specifies how the source and target currencies are to be arranged using the reserved keywords \in and \out to refer to the source and target currencies, respectively. In the example below the first line implements a format close to the default, the second line displays the result of the conversion, and the third line does not show the conversion at all (although it happens behind the scenes). The latter is useful if you want to make use of the currency formatting features of the package without being interested in any conversion.

1 000 DM (= 3 353,85 FRF)
3 353,85 FRF
1 000 DM

```
\usepackage{euro}
\EUROFORMAT{main}{\in\ (=\,\out)} \EURO{DEM}[FRF]{1000}\par
\EUROFORMAT{main}{\out}           \EURO{DEM}[FRF]{1000}\par
\EUROFORMAT{main}{\in}            \EURO{DEM}     {1000}
```

3-1-39

The in and out formats specify how the source and target currencies should be formatted using the reserved keywords \val (monetary amount), \iso (currency code), and \sym (currency symbol if defined; ISO code otherwise).

DM 1 000 (FRF 3 353,85)

```
\usepackage{euro}
\EUROFORMAT{in}{\sym~\val} \EUROFORMAT{out}{\iso~\val}
\EURO{DEM}[FRF]{1000}
```

3-1-40

Perhaps more interesting are the possibilities to influence the formatting of monetary amounts, for which the package offers five declarations to be used in the second argument to \EUROFORMAT. The \round declaration specifies where to round the monetary amount: positive values round to the integer digits and negative values to the fractional digits. For example, \round{-3} means show and round to three fractional digits. The \form declaration takes three arguments: the integer group separator (default \,), the decimal separator (default a comma), and the fractional group separator (default \,).

The first argument can be either `all` to define the default number formatting or an ISO currency code to modify the formatting for a single currency.

1,022·5838 Euro
−335·3855 FRF
9,900,000 Lit.

3-1-41

```
\usepackage{euro} \EUROFORMAT{main}{\out}
\EUROFORMAT{all}{\round{-4}\form{,}{\textperiodcentered}{}}
\EUROFORMAT{ITL}{\round{2}}
\noindent \EURO{DEM}{2000}\\ \EURO{DEM}[FRF]{-100}\\
\EURO{DEM}[ITL]{10000}
```

The `\minus` declaration formats negative values by executing its first argument before the number and its second argument after it (default `\minus{$-$}{}`). The number itself is typeset unsigned, so that a minus sign has to be supplied by the declaration. The `\plus` declaration is the analogue for dealing with positive numbers (default `\plus{}{}`).

3-1-42

+1 022,58 Euro −335,39 FRF

```
\usepackage{color,euro} \EUROFORMAT{main}{\out}
\EUROFORMAT{all}{\plus{$+$}{}\minus{\color{blue}$-$}{}}
\EURO{DEM}{2000}\quad \EURO{DEM}[FRF]{-100}
```

The `\zero` declaration takes three arguments to describe what to do if everything is zero, the integer part is zero, or the fractional part is zero. In the first and third arguments, the decimal separator has to be entered as well, so it should correspond to the default or the value given in the `\form` command.

3-1-43

0,00 € 0,51 € 1,– €

```
\usepackage{eurosans,euro}
\EUROFORMAT{main}{\out} \EUROSYM{EUR}{\euro}
\EUROFORMAT{all}{\zero{0,00}{0}{,--}}
\EURO{DEM}{0}\quad \EURO{DEM}{1}\quad \EURO{EUR}{1}
```

3.1.10 lettrine—Dropping your capital

In certain types of publications you may find the first letter of some paragraphs being highlighted by means of an enlarged letter often dropped into the paragraph body (so that the paragraph text flows around it) and usually followed by the first phrase or sentence being typeset in a special font. Applications range from chapter openings in novels, or indications of new thoughts in the text, to merely decorative elements to produce lively pages in a magazine. This custom can be traced back to the early days of printing, when such initials were often hand-colored after the printing process was finished. It originates in the manuscripts of the Middle Ages; that is, it predates the invention of printing.

`\lettrine[`*key/val-list*`]{`*initial*`}{`*text*`}`

The package lettrine written by Daniel Flipo lets you create such initials by providing the command `\lettrine`. In its simplest form it takes two arguments: the letter to become an initial and the follow-up text to be typeset in a special font, by default in \scshape.

LA MOITIÉ DES PASSAGERS, affaiblis, expirants de ces angoisses inconcevables que le roulis d’un vaisseau porte dans les nerfs et dans toutes les humeurs du corps agitées en sens contraire, . . .

```
\usepackage{lettrine} \usepackage[latin1]{inputenc}
\usepackage[french]{babel}
\lettrine{L}{a moitié des passagers,} affaiblis,
expirants de ces angoisses inconcevables que le
roulis d’un vaisseau porte dans les nerfs et
dans toutes les humeurs du corps agitées en sens
contraire, \ldots
```

3-1-44

The font used for the initial is, by default, a larger size of the current text font. Alternatively, you can specify a special font family by redefining the command \LettrineFontHook using standard NFSS commands. Similarly, the font used for the text in the second argument can be modified by changing \LettrineTextFont.

Because the \lettrine command calculates the initial size to fit a certain number of lines, you need scalable fonts to obtain the best results. As the examples in this book are typeset in Adobe Times and Helvetica by default, we have no problems here. Later examples use Palatino, which is also a scalable Type 1 font. But if you use a bitmapped font, such as Computer Modern, you might have to use special .fd files (see Chapter 7, pages 419ff) to achieve acceptable results.

LA MOITIÉ DES PASSAGERS, affaiblis, expirants de ces angoisses inconcevables que le roulis d’un vaisseau porte dans les nerfs et dans toutes les humeurs du corps agitées en sens contraire, . . .

```
\usepackage{lettrine} \usepackage[latin1]{inputenc}
\usepackage[french]{babel}
\renewcommand\LettrineFontHook{\sffamily\bfseries}
\renewcommand\LettrineTextFont{\sffamily\scshape}
\lettrine{L}{a moitié des passagers,} affaiblis,
expirants de ces angoisses inconcevables que le
roulis d’un vaisseau porte dans les nerfs et
dans toutes les humeurs du corps agitées en sens
contraire, \ldots
```

3-1-45

Many books on typography give recommendations about how to best set large initials with respect to surrounding text. For highest quality it is often necessary to manually adjust the placement depending on the shape of the initial. For example, it is often suggested that letters with a projecting left stem should overhang into the margin. The \lettrine command caters to this need by supporting an optional argument in which you can specify adjustments in the form of a comma-separated list of key/value pairs.

The size of the initial is calculated by default to have a height of two text lines (stored in \DefaultLines); with the keyword lines you can change this value to a different number of lines. There is an exception: if you specify lines=1 the initial is still made two lines high, but instead of being dropped is placed onto the baseline of the first text line.

If you want a dropped initial that also extends above the first line of text, then use the keyword loversize. A value of .2 would enlarge the initial by 20%. The default value for this keyword is stored in \DefaultLoversize. This keyword is also useful in conjunction with lraise (default 0 in \DefaultLraise). In case of an initial with a large descender such as a “Q” you may have to raise the initial to avoid it overprinting following lines. In that case loversize can be used to reduce the height so as to align the initial properly.

With the keyword lhang you specify how much the initial extends into the margin. The value is specified as a fraction—that is, between 0 and 1. Its document default is stored in \DefaultLhang.

QUAND ILS FURENT revenus un peu à eux, ils marchèrent vers Lisbonne ; il leur restait quelque argent, avec lequel ils espéraient se sauver de la faim après avoir échappé à la tempête . . .

3-1-46

```
\usepackage{palatino,lettrine}
\usepackage[latin1]{inputenc}
\usepackage[french]{babel}

\lettrine[lines=3, loversize=-0.1, lraise=0.1,
  lhang=.2]{Q}{uand ils furent} revenus un peu à eux,
ils marchèrent vers Lisbonne ; il leur restait quelque
argent, avec lequel ils espéraient se sauver de la
faim après avoir échappé à la tempête \ldots
```

The distance between the initial and the following text in the first line is controlled by the command \DefaultFindent (default 0pt) and can be overwritten using the keyword findent. The indentation of following lines is by default 0.5em (stored in \DefaultNindent) but can be changed through the keyword nindent. If you want to specify a sloped indentation you can use the keyword slope, which applies from the third line onward. Again the default value can be changed via the command \DefaultSlope, though it seems questionable that you would ever want anything different than 0pt since a slope is normally only used for letters like “A” or “V”.

À PEINE ONT-ILS MIS le pied dans la ville en pleurant la mort de leur bienfaiteur, qu’ils sentent la terre trembler sous leurs pas ; . . .

3-1-47

```
\usepackage{palatino,lettrine}
\usepackage[latin1]{inputenc}
\usepackage[french]{babel}

\lettrine[lines=4, slope=0.6em, findent=-1em,
   nindent=0.6em]{À} { peine ont-ils mis} le pied dans
la ville en pleurant la mort de leur bienfaiteur,
qu’ils sentent la terre trembler sous leurs pas; \ldots
```

The example above clearly demonstrates that the size calculation for the initial does not take accents into account, which is normally the desired behavior. It is nevertheless possible to manually adjust the size using loversize.

To attach material to the left of the initial, such as some opening quote, you can use the keyword ante. It is the only keyword for which no command exists to set the default.

By modifying the default settings you can easily adapt the package to typeset initials the way you like. This can be done either in the preamble or in a file with the name lettrine.cfg, which is loaded if found.

3.1.11 Paragraph justification in LaTeX

For formatting paragraphs LaTeX deploys the algorithms already built into the TeX program, which by default produce justified paragraphs. In other words, spaces between words will be slightly stretched or shortened to produce lines of equal length. TeX achieves this outcome with an algorithm that attempts to find an optimal solution for a whole paragraph, using the current settings of about 20 internal parameters. They include aspects such as trying to produce visually compatible lines, such that a tight line is not followed by one very loosely typeset, or considering several hyphens in a row as a sign of bad quality. The interactions between these parameters are very subtle and even experts find it difficult to predict the results when tweaking them. Because the standard settings are suitable for nearly all applications, we describe only some of the parameters in this book. Appendix B.3.3 discusses how to trace the algorithm. If you are interested in delving further into the matter of automatic paragraph breaking, refer to *The TeXbook* [82, chap. 14], which describes the algorithm in great of detail, or to the very interesting article by Michael Plass and Donald Knuth on the subject, which is reprinted in [98].

The downside of the global optimizing approach of TeX, which you will encounter sooner or later, is that making small changes, like correcting a typo near the end of a paragraph, can have drastic and surprising effects, as it might affect the line breaking of the whole paragraph. It is possible, and not even unlikely, that, for example, the *removal* of a word might actually result in making a paragraph one line *longer*. This behavior can be very annoying if you are near the end of finishing an important project (like the second edition of this book) and a correction wreaks havoc on your already manually adjusted page breaks. In such a situation it is best to place `\linebreak` or `\pagebreak` commands into strategic places to force TeX to choose a solution that it would normally consider inferior. To be able to later get rid of such manual corrections you can easily define your own commands, such as

```
\newcommand\finallinebreak{\linebreak}
```

rather than using the standard LaTeX commands directly. This helps you to distinguish the layout adjustments for a particular version from other usages of the original commands—a method successfully used in the preparation of this book.

The interword spacing in a justified paragraph (the white space between individual words) is controlled by several TeX parameters—the most important ones are `\tolerance` and `\emergencystretch`. By setting them suitably for your document you can prevent most or all of the "Overfull box" messages without any manual line breaks. The `\tolerance` command is a means for setting how much the interword space in a paragraph is allowed to diverge from its optimum value.[1] This command is a TeX (not LaTeX) counter and therefore it has an uncommon

[1] The optimum is font defined; see Section 7.10.3 on page 428.

assignment syntax—for example, `\tolerance=500`. Lower values make TeX try harder to stay near the optimum; higher values allow for loose typesetting. The default value is often 200. When TeX is unable to stay in the given tolerance you will find overfull boxes in your output (i.e., lines sticking out into the margin like this). Enlarging the value of `\tolerance` means that TeX will also consider poorer but still acceptable line breaks, instead of turning the problem over to you for manual intervention. Sensible values are between 50 and 9999. Do not use 10000 or higher, as it allows TeX to produce arbitrary bad lines (like this one). If you really need fully automated line breaking, it is better to set the length parameter `\emergencystretch` to a positive value. If TeX cannot break a paragraph without producing overfull boxes (due to the setting of `\tolerance`) and `\emergencystretch` is positive, it will add this length as stretchable space to every line, thereby accepting line-breaking solutions that have been rejected before. You may get some underfull box messages because all the lines are now set in a loose measure, but this result will still look better than a single horrible line in the middle of an otherwise perfectly typeset paragraph.

Careful with TeX's idea about infinitely bad

LaTeX has two predefined commands influencing the above parameters: `\fussy`, which is the default, and `\sloppy`, which allows for relatively bad lines. The `\sloppy` command is automatically applied by LaTeX in some situations (e.g., when typesetting `\marginpar` arguments or `p` columns in a `tabular` environment) where perfect line breaking is seldom possible due to the narrow measure.

Unjustified text

While the theory on producing high-quality justified text is well understood (even though surprisingly few typesetting systems other than TeX use algorithms that can produce high quality other than by chance), the same cannot be said for the situation when unjustified text is being requested. This may sound strange at first hearing. After all, why should it be difficult to break a paragraph into lines of different length? The answer lies in the fact that we do not have quantifiable quality measures that allow us to easily determine whether a certain breaking is good or bad. In comparison to its work with justified text, TeX does a very poor job when asked to produce unjustified paragraphs. Thus, to obtain the highest quality we have to be prepared to help TeX far more often by adding explicit line breaks in strategic places. A good introduction to the problems in this area is given in an article by Paul Stiff [154].

The main type of unjustified text is the one in which lines are set flush left but are unjustified at the right. For this arrangement LaTeX offers the environment `flushleft`. It typesets all text in its scope "flush left" by adding very stretchable white space at the right of each line; that is, it sets the internal parameter `\rightskip` to `0pt plus 1fil`. This setting often produces very ragged-looking paragraphs as it makes all lines equally good independent of the amount of text they contain. In addition, hyphenation is essentially disabled because a hyphen adds to the “badness” of a line and, as there is nothing to counteract it, TEX’s paragraph-breaking algorithm will normally choose line breaks that avoid them.

“The LATEX document preparation
system is a special version of
Donald Knuth’s TEX program.
TEX is a sophisticated program
designed to produce high-quality
typesetting, especially for
mathematical text.”

```
\begin{flushleft}
``The \LaTeX{} document preparation system is a special
version of Donald Knuth's \TeX{} program.  \TeX{} is a
sophisticated program designed to produce high-quality
typesetting, especially for mathematical text.''
\end{flushleft}
```

3-1-48

In summary, LATEX’s flushleft environment is not particularly well suited to continuous unjustified text, which should vary at the right-hand boundary only to a certain extent and where appropriate should use hyphenation (see the next section for alternatives). Nevertheless, it can be useful to place individual objects, like a graphic, flush left to the margin, especially since this environment adds space above and below itself in the same way as list environments do.

Another important restriction is the fact that the settings chosen by this environment have no universal effect, because some environments (e.g., minipage or tabular) and commands (e.g., \parbox, \footnote, and \caption) restore the alignment of paragraphs to full justification. That is, they set the \rightskip length parameter to 0pt and thus cancel the stretchable space at the right line endings. A way to automatically deal with this problem is provided by the package ragged2e (see next section).

Other ways of typesetting paragraphs are flush right and centered, with the flushright and center environments, respectively. In these cases the line breaks are usually indicated with the \\ command, whereas for ragged-right text (the flushleft environment discussed above) you can let LATEX do the line breaking itself (if you are happy with the resulting quality).

The three environments discussed in this section work by changing declarations that control how TEX typesets paragraphs. These declarations are also available as LATEX commands, as shown in the following table of correspondence:

environment:	center	flushleft	flushright
command:	\centering	\raggedright	\raggedleft

The commands neither start a new paragraph nor add vertical space, unlike the corresponding environments. Hence, the commands can be used inside other environments and inside a \parbox, in particular, to control the alignment in p columns of an array or tabular environment. Note, however, that if they are used in the last column of a tabular or array environment, the \\ is no longer available to denote the end of a row. Instead, the command \tabularnewline can be used for this purpose (see also Section 5.2.1).

3.1.12 ragged2e—Enhancing justification

The previous subsection discussed the deficiencies of LaTeX's `flushleft` and `flushright` environments. The package `ragged2e` written by Martin Schröder sets out to provide alternatives that do not produce such extreme raggedness. This venture is not quite as simple as it sounds, because it is not enough to set `\rightskip` to something like `0pt plus 2em`. Notwithstanding the fact that this would result in TeX trying hard to keep the line endings within the `2em` boundary, there remains a subtle problem: by default, the interword space is also stretchable for most fonts. Thus, if `\rightskip` has only finite stretchability, TeX will distribute excess space equally to all spaces. As a result, the interword spaces will have different width, depending on the amount of material in the line. The solution is to redefine the interword space so that it no longer can stretch or shrink by specifying a suitable (font-dependent) value for `\spaceskip`. This internal TeX parameter, if nonzero, represents the current interword space, overwriting the default that is defined by the current font.

By default, the package does not modify the standard LaTeX commands and environments discussed in the previous section, but instead defines its own using the same names except that some letters are uppercased.[1] The new environments and commands are given in the following correspondence table:

environment:	`Center`	`FlushLeft`	`FlushRight`
command:	`\Centering`	`\RaggedRight`	`\RaggedLeft`

They differ from their counterparts of the previous section not only in the fact that they try to produce less ragged output, but also in their attempt to provide additional flexibility by easily letting you change most of their typesetting aspects.

Overloading the original commands

As typesetting the mixed-case commands and environments is somewhat tedious, you can overload the original commands and environments, such as `\raggedright`, with the new definitions by supplying the `newcommands` option when loading the package.

The package offers a large number of parameters to define the exact behavior of the new commands and environments (see Table 3.2 on the next page). For `\RaggedRight` or `FlushLeft` the white space added at the right of each line can be specified as `\RaggedRightRightskip`, the one at the left can be specified as `\RaggedRightLeftskip`, the paragraph indentation to use is available as `\RaggedRightParindent`, and even the space added to fill the last line is available as `\RaggedRightParfillskip`. Similarly, the settings for `\Centering` and `\RaggedLeft` can be altered; just replace `RaggedRight` in the parameter names with either `Centering` or `RaggedLeft`.

Unjustified setting as the default

To set a whole document unjustified, specify `document` as an option to the `ragged2e` package. For the purpose of justifying individual paragraphs the

[1] This is actually against standard naming conventions. In most packages mixed-case commands indicate interface commands to be used by designers in class files or in the preamble, but not commands to be used inside documents.

`\RaggedLeftParindent`	`0pt`	`\RaggedLeftLeftskip`	`0pt plus 2em`
`\RaggedLeftRightskip`	`0pt`	`\RaggedLeftParfillskip`	`0pt`
`\CenteringParindent`	`0pt`	`\CenteringLeftskip`	`0pt plus 2em`
`\CenteringRightskip`	`0pt plus 2em`	`\CenteringParfillskip`	`0pt`
`\RaggedRightParindent`	`0pt`	`\RaggedRightLeftskip`	`0pt`
`\RaggedRightRightskip`	`0pt plus 2em`	`\RaggedRightParfillskip`	`0pt plus 1fil`
`\JustifyingParindent`	`1 em`	`\JustifyingParfillskip`	`0pt plus 1fil`

Table 3.2: Parameters used by `ragged2e`

package offers the command `\justifying` and the environment `justify`. Both can be customized using the length parameters `\JustifyingParindent` and `\JustifyingParfillskip`.

Thus, to produce a document with a moderate amount of raggedness and paragraphs indented by 12 pt, you could use a setting like the one in the following example (compare it to Example 3-1-48 on page 104).

"The LaTeX document preparation system is a special version of Donald Knuth's TeX program. TeX is a sophisticated program designed to produce high-quality typesetting, especially for mathematical text."

```
\usepackage[document]{ragged2e}
\setlength\RaggedRightRightskip{0pt plus 1cm}
\setlength\RaggedRightParindent{12pt}
``The \LaTeX{} document preparation system is a special
version of Donald Knuth's \TeX{} program.  \TeX{} is a
sophisticated program designed to produce high-quality
typesetting, especially for mathematical text.''
```

3-1-49

Unjustified settings in narrow columns

In places with narrow measures (e.g., `\marginpar`s, `\parbox`es, `minipage` environments, or p-columns of `tabular` environments), the justified setting usually produces inferior results. With the option `raggedrightboxes`, paragraphs in such places are automatically typeset using `\RaggedRight`. If necessary, `\justifying` can be used to force a justified paragraph in individual cases.

The default values

The use of `em` values in the defaults (see Table 3.2) means that special care is needed when loading the package, as the `em` is turned into a real dimension at this point! The package should therefore be loaded *after* the body font and size have been established—for example, after font packages have been loaded.

Instead of using the defaults listed in Table 3.2, one can instruct the package to mimic the original LaTeX definitions by loading it with the option `originalparameters` and then changing the parameter values as desired.

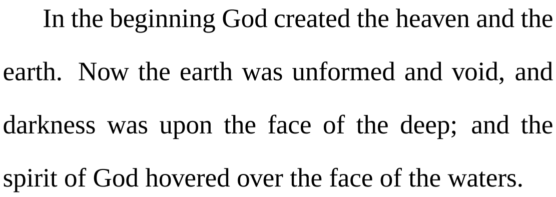
3.1.13 `setspace`—Changing interline spacing

The `\baselineskip` command is TeX's parameter for defining the *leading* (normal vertical distance) between consecutive baselines. Standard LaTeX defines a leading approximately 20% larger than the design size of the font (see Section 7.9.1 on page 413). Because it is not recommended to change the setting of \baselineskip directly, LaTeX provides the \baselinestretch command to allow for changing \baselineskip at all sizes globally.

Be aware that after the \renewcommand{\baselinestretch}{1.5} command is issued, the leading will not increase immediately. A font size changing command (e.g., \small, \Large) must be executed to make the new value take effect.

The package setspace (by Geoffrey Tobin and others) provides commands and environments for typesetting with variable spacing (primarily double and one-and-a-half). Three commands—\singlespacing, \onehalfspacing, and \doublespacing—are available for use in the preamble to set the overall spacing for the document. Alternatively, a different spacing value can be defined by placing a \setstretch command in the preamble. It takes the desired spacing factor as a mandatory argument. In the absence of any of the above commands, the default setting is single spacing.

To change the spacing inside a document three specific environments—singlespace, onehalfspace, and doublespace—are provided. They set the spacing to single (default), one-and-a-half, and double spacing, respectively. These environments cannot be nested.

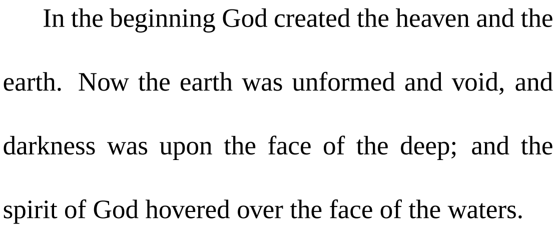


In the beginning God created the heaven and the earth. Now the earth was unformed and void, and darkness was upon the face of the deep; and the spirit of God hovered over the face of the waters.

3-1-50

```
\usepackage{setspace}
\begin{doublespace}
 In the beginning God created the heaven
 and the earth. Now the earth was unformed
 and void, and darkness was upon the face
 of the deep; and the spirit of God
 hovered over the face of the waters.
\end{doublespace}
```

For any other spacing values the generic environment spacing should be used. Its mandatory parameter is the value of \baselinestretch for the text enclosed by the environment.



In the beginning God created the heaven and the earth. Now the earth was unformed and void, and darkness was upon the face of the deep; and the spirit of God hovered over the face of the waters.

3-1-51

```
\usepackage{setspace}
\begin{spacing}{2.0}
 In the beginning God created the heaven
 and the earth. Now the earth was unformed
 and void, and darkness was upon the face
 of the deep; and the spirit of God
 hovered over the face of the waters.
\end{spacing}
```

In the above example the coefficient “2.0” produces a larger leading than the “double spacing” (doublespace environment) required for some publications. With the spacing environment the leading is increased twice—once by \baselineskip (where LaTeX already adds about 20% space between baselines) and a second time by setting \baselinestretch. “Double spacing” means that the vertical distance between baselines is about twice as large as the font size.

spacing	10 pt	11 pt	12 pt
one and one-half	1.25	1.21	1.24
double	1.67	1.62	1.66

Table 3.3: Effective `\baselinestretch` values for different font sizes

Since `\baselinestretch` refers to the ratio between the desired distance and the `\baselineskip`, the values of `\baselinestretch` for different document base font sizes (and at two different optical spacings) can be calculated and are presented in Table 3.3.

3.1.14 picinpar—Making rectangular holes

The package picinpar (created by Friedhelm Sowa based on earlier work by Alan Hoenig) allows "windows" to be typeset inside paragraphs. The basic environment is `window`. It takes one mandatory argument specified in contrast to LaTeX conventions in square brackets, in the form of a comma-separated list of four elements. These elements are the number of lines before the window starts; the alignment of the window inside the paragraph (`l` for left, `c` for centered, and `r` for right); the material shown in the window; and explanatory text about the contents in the window (e.g., the caption).

In this case we center a word printed vertically inside the paragraph. It is not difficult to understand that tables can also be easily included with the `tabwindow` environment.

When a paragraph ends, like here, and the window is not yet finished, then it just continues past the paragraph boundary, right into the next one(s).

H
e
l
l
o

```
\usepackage{picinpar}
\begin{window}[1,c,%
   \fbox{\shortstack{H\\e\\l\\l\\o}},]
 In this case we center a word printed
 vertically inside the paragraph. It is not
 difficult to understand that tables can also
 be easily included with the \texttt{tabwindow}
 environment.\par When a paragraph ends, like
 here, and the window is not yet finished,
 then it just continues past the paragraph
 boundary, right into the next one(s).
\end{window}
```

3-1-52

If you look at the above example you will notice that the second paragraph is not properly indented. You can fix this defect by requesting an explicit indentation using `\par\indent`, if necessary.

Centering a window as in the previous example works only if the remaining text width on either side is still suitably wide (where "suitably" means larger than one inch). Otherwise, the package will simply fill it with white space.

The package also provides two variant environments, `figwindow` and `tabwindow`. They can format the explanatory text as a caption, by adding a caption number. You should, however, be careful when mixing such "nonfloating" floats with standard figure or table environments, because the latter might get deferred and this way mess up the numbering of floats.

The next example shows such an embedded figure—a map of Great Britain placed inside a paragraph. Unfortunately, the caption formatting is more or less hard-wired into the package; if you want to change it, you have to modify an internal command named \@makewincaption.

Figure 1: United Kingdom

Is this a dagger which I see before me, The handle toward my hand? Come, let me clutch thee. I have thee not, and yet I see thee still. Art thou not, fatal vision, sensible To feeling as to sight? or art thou but A dagger of the mind, a false creation, Proceeding from the heat-oppressed brain? I see thee yet, in form as palpable As this which now I draw. Thou marshall'st me the way that I was going; And such an instrument I was to use. Mine eyes are made the fools o' the other senses, Or else worth all the rest; I see thee still, And on thy blade and dudgeon gouts of blood, Which was not so before. (*Macbeth*, Act II, Scene 1). 3-1-53

```
\usepackage{picinpar,graphicx}
\begin{figwindow}[3,l,%
 \fbox{\includegraphics[width=30mm]{ukmap}},%
                  {United Kingdom}]
 Is this a dagger which I see before me, The
 handle toward my hand? Come, let me clutch
 thee. I have thee not, and yet I see thee
 still.  Art thou not, fatal vision,
 sensible To feeling as to sight?  or art
 thou but A dagger of the mind, a false
 creation, Proceeding from the
 heat-oppressed brain?  I see thee yet, in
 form as palpable As this which now I draw.
 Thou marshall'st me the way that I was
 going; And such an instrument I was to use.
 Mine eyes are made the fools o' the other
 senses, Or else worth all the rest; I see
 thee still, And on thy blade and dudgeon
 gouts of blood, Which was not so before.
 (\emph{Macbeth}, Act II, Scene 1).
\end{figwindow}
```

3.2 Footnotes, endnotes, and marginals

LaTeX has facilities to typeset "inserted" text, such as marginal notes, footnotes, figures, and tables. The present section looks more closely at different kinds of notes, while Chapter 6 describes floats in more detail.

We start by discussing the possibilities offered through standard LaTeX's footnote commands and explain how (far) they can be customized. For two-column documents, a special layout for footnotes is provided by the ftnright package, which moves all footnotes to the bottom of the right column. This is followed by a presentation of the footmisc package, which overcomes most of the limitations of the standard commands and offers a wealth of additional features. The manyfoot package (which can be combined with footmisc) extends the footnote support for disciplines like linguistics by providing several independent footnote commands.

Support for endnotes is provided through the package endnotes, which allows for mixing footnotes and endnotes and can also be used to provide chapter notes, as required by some publishers. The section concludes with a discussion of marginal notes, which are already provided by standard LaTeX.

3.2.1 Using standard footnotes

A sharp distinction is made between footnotes in the main text and footnotes inside a `minipage` environment. The former are numbered using the `footnote` counter, while inside a `minipage` the `\footnote` command is redefined to use the `mpfootnote` counter. Thus, the representation of the footnote mark is obtained by the `\thefootnote` or the `\thempfootnote` command depending on the context. By default, it typesets an Arabic number in text and a lowercase letter inside a `minipage` environment. You can redefine these commands to get a different representation by specifying, for example, footnote symbols, as shown in the next example.

text text text[*] text text[†] text.

[*] The first
[†] The second

```
\renewcommand\thefootnote
             {\fnsymbol{footnote}}
text text text\footnote{The first}
text text\footnote{The second} text.
```

3-2-1

Peculiarities inside a `minipage`

Footnotes produced with the `\footnote` command inside a `minipage` environment use the `mpfootnote` counter and are typeset at the bottom of the parbox produced by the `minipage`. However, if you use the `\footnotemark` command in a `minipage`, it will produce a footnote mark in the same style and sequence as the main text footnotes—that is, stepping the `footnote` counter and using the `\thefootnote` command for the representation. This behavior allows you to produce a footnote inside your `minipage` that is typeset in sequence with the main text footnotes at the bottom of the page: you place a `\footnotemark` inside the `minipage` and the corresponding `\footnotetext` after it.

… main text …

Footnotes in a minipage are numbered using lowercase letters.[a]
This text references a footnote at the bottom of the page.[1] And another[b] note.

[a] Inside minipage
[b] Inside again

[1] At bottom of page

```
\noindent\ldots{} main text \ldots
\begin{center}
 \begin{minipage}{.7\linewidth}
  Footnotes in a minipage are numbered using
  lowercase letters.\footnote{Inside minipage}
  \par This text references a footnote at the
  bottom of the page.\footnotemark{}
  And another\footnote{Inside again} note.
 \end{minipage}\footnotetext{At bottom of page}
\end{center}
\ldots{} main text \ldots
```

3-2-2

As the previous example shows, if you need to reference a `minipage` footnote several times, you cannot use `\footnotemark` because it refers to footnotes typeset at the bottom of the page. You can, however, load the package footmisc and then use \mpfootnotemark in place of \footnotemark. Just like \footnotemark, the \mpfootnotemark command first increments its counter and then displays its value. Thus, to refer to the previous value you typically have to decrement it first, as shown in the next example.

Main text . . .

Footnotes in a minipage are numbered using lowercase letters.[a] This text references the previous footnote.[a] And another[b] note.

[a] Inside minipage
[b] Inside as well

```
\usepackage{footmisc}
\noindent Main text \ldots \begin{center}
 \begin{minipage}{.7\linewidth}
 Footnotes in a minipage are numbered using
 lowercase letters.\footnote{Inside minipage}
 \par This text references the previous
 footnote.\addtocounter{mpfootnote}{-1}%
          \mpfootnotemark{}
 And another\footnote{Inside as well} note.
 \end{minipage}
\end{center} \ldots{} main text \ldots
```

3-2-3

LaTeX does not allow you to use a \footnote inside another \footnote command, as is common in some disciplines. You can, however, use the \footnotemark command inside the first footnote and then put the text of the footnote's footnote as the argument of a \footnotetext command. For other special footnote requirements consider using the manyfoot package (described below).

Some[1] text and some more text.

[1] A sample[2] footnote.
[2] A subfootnote.

```
Some\footnote{A sample\footnotemark{}
footnote.}\footnotetext{A subfootnote.}
text and some more text.
```

3-2-4

What if you want to reference a given footnote? You can use LaTeX's normal \label and \ref mechanism, although you may want to define your own command to typeset the reference in a special way. For instance:

This is some text.[1]
. . . as shown in footnote (1) on page 6,. . .

[1] Text inside referenced footnote.

```
\newcommand\fnref[1]{\unskip~(\ref{#1})}
This is some text.\footnote{Text inside
referenced footnote\label{fn:myfoot}.}\par
\ldots as shown in footnote\fnref{fn:myfoot}
on page~\pageref{fn:myfoot},\ldots
```

3-2-5

Standard LaTeX does not allow you to construct footnotes inside tabular material. Section 5.8 describes several ways of tackling that problem.

3.2.2 Customizing standard footnotes

Footnotes in LaTeX are generally simple to use and provide a quite powerful mechanism to typeset material at the bottom of a page.[1] This material can consist of several paragraphs and can include lists, inline or display mathematics, tabular material, and so on.

LaTeX offers several parameters to customize footnotes. They are shown schematically in Figure 3.1 on the next page and are described below:

`\footnotesize` The font size used inside footnotes (see also Table 7.1 on page 342).

`\footnotesep` The height of a strut placed at the beginning of every footnote. If it is greater than the `\baselineskip` used for `\footnotesize`, then additional vertical space will be inserted above each footnote. See Appendix A.2.3 for more information about struts.

`\skip\footins` A low-level TeX length parameter that defines the space between the main text and the start of the footnotes. You can change its value with the `\setlength` or `\addtolength` command by putting `\skip\footins` into the first argument:

```
\addtolength{\skip\footins}{10mm plus 2mm}
```

`\footnoterule` A macro to draw the rule separating footnotes from the main text that is executed right after the vertical space of `\skip\footins`. It should take zero vertical space; that is, it should use a negative skip to compensate for any positive space it occupies. The default definition is equivalent to the following:

```
\renewcommand\footnoterule{\vspace*{-3pt}%
   \hrule width 2in height 0.4pt \vspace*{2.6pt}}
```

Note that TeX's `\hrule` command and not LaTeX's `\rule` command is used. Because the latter starts a paragraph, it would be difficult to calculate the spaces needed to achieve a net effect of zero height. For this reason producing a fancier "rule" is perhaps best done by using a zero-sized picture environment to position the rule object without actually adding vertical space.

In the report and book classes, footnotes are numbered inside chapters; in article, footnotes are numbered sequentially throughout the document. You can change the latter default by using the `\@addtoreset` command (see Appendix A.1.4). However, do not try to number your footnotes within pages with

[1] An interesting and complete discussion of this subject appeared in the French TeX Users' Group magazine *Cahiers GUTenberg* [10, 133].

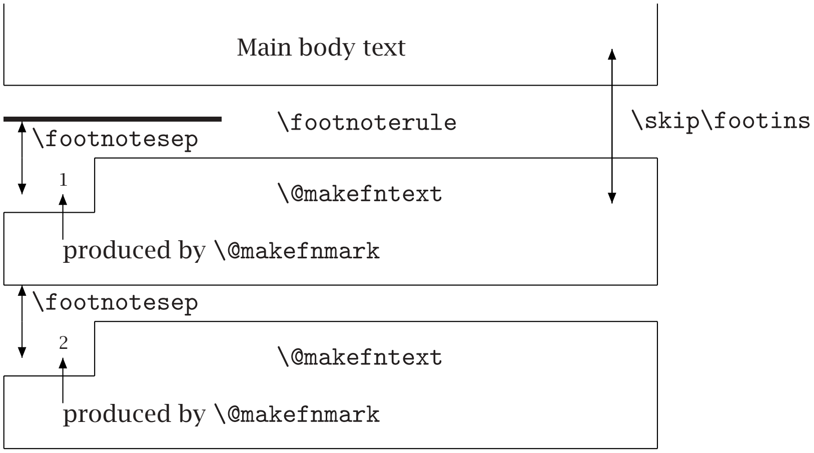


Figure 3.1: Schematic layout of footnotes

the help of this mechanism. LaTeX is looking ahead while producing the final pages, so your footnotes would most certainly be numbered incorrectly. To number footnotes on a per-page basis, use the footmisc or perpage package (described below).

The command `\@makefnmark` is normally used to generate the footnote mark. One would expect this command to take one argument (the current footnote number), but in fact it takes none. Instead, it uses the command `\@thefnmark` to indirectly refer to that number. The reason is that depending on the position (inside or outside of a `minipage`) a different counter needs to be accessed. The definition, which by default produces a superscript mark, looks roughly as follows:

```
\renewcommand\@makefnmark
             {\mbox{\textsuperscript{\normalfont\@thefnmark}}}
```

The `\footnote` command executes `\@makefntext` inside a `\parbox`, with a width of `\columnwidth`. The default version looks something like:

```
\newcommand\@makefntext[1]
           {\noindent\makebox[1.8em][r]{\@makefnmark}#1}
```

This will place the footnote mark right aligned into a box of width `1.8em` directly followed by the footnote text. Note that it reuses the `\@makefnmark` macro, so any change to it will, by default, modify the display of the mark in both places. If you want the text set flush left with the number placed into the margin, then you could use the redefinition shown in the next example. Here we do not use `\@makefnmark` to format the mark, but rather access the number via `\@thefnmark`. As a result, the mark is placed onto the baseline instead of being raised. Thus, the marks in the text and at the bottom are formatted differently.

text text text[1] text text[2] text.

1. The first
2. The second

```
\makeatletter
\renewcommand\@makefntext[1]%
   {\noindent\makebox[0pt][r]{\@thefnmark.\,}#1}
\makeatother
text text text\footnote{The first}
text text\footnote{The second} text.
```

3-2-6

3.2.3 ftnright—Right footnotes in a two-column environment

It is sometimes desirable to group all footnotes in a two-column document at the bottom of the right column. This can be achieved by specifying the ftnright package written by Frank Mittelbach. The effect of this package is shown in Figure 3.2 on the facing page—the first page of the original documentation (including its spelling errors) of the ftnright implementation. It is clearly shown how the various footnotes collect in the lower part of the right-hand column.

The main idea for the ftnright package is to assemble the footnotes of all columns on a page and place them all together at the bottom of the right column. The layout produced allows for enough space between footnotes and text and, in addition, sets the footnotes in smaller type.[1] Furthermore, the footnote markers are placed at the baseline instead of raising them as superscripts.[2]

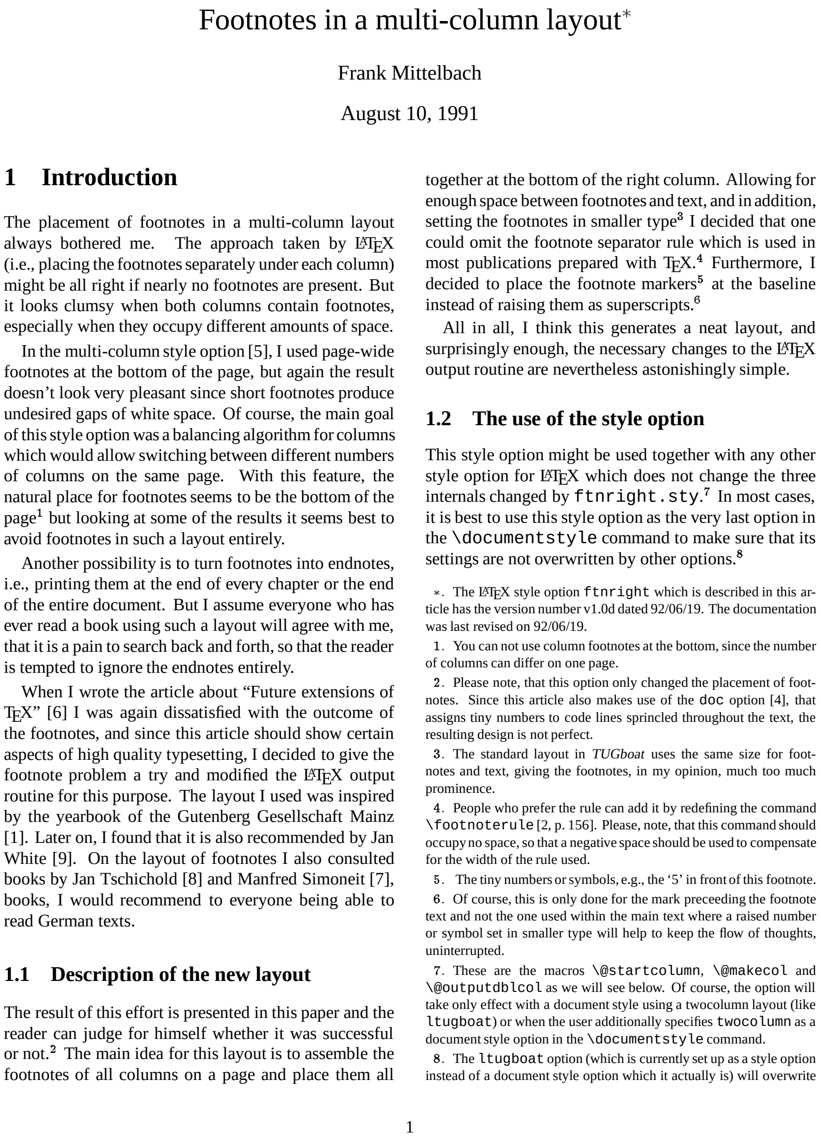
This package can be used together with most other class files for LaTeX. Of course, the ftnright package will take effect only with a document using a two-column layout specified with the `twocolumn` option on the `\documentclass` command. In most cases, it is best to use ftnright as the very last package to make sure that its settings are not overwritten by other options.

3.2.4 footmisc—Various footnotes styles

Since standard LaTeX offers only one type of footnotes and only limited (and somewhat low-level) support for customization, several people developed small packages that provided features otherwise not available. Many of these earlier efforts were captured by Robin Fairbairns in his footmisc package, which supports, among other things, page-wise numbering of footnotes and footnotes formatted as a single paragraph at the bottom of the page. In this section we describe the features provided by this package, showing which packages it supersedes whenever applicable.

[1] Some journals use the same size for footnotes and text, which sometimes makes it difficult to distinguish footnotes from the main text.

[2] Of course, this is done only for the mark preceding the footnote text and not the one used within the main text, where a raised number or symbol set in smaller type will help to keep the flow of thoughts uninterrupted.



Footnotes in a multi-column layout[*]

Frank Mittelbach

August 10, 1991

1 Introduction

The placement of footnotes in a multi-column layout always bothered me. The approach taken by LaTeX (i.e., placing the footnotes separately under each column) might be all right if nearly no footnotes are present. But it looks clumsy when both columns contain footnotes, especially when they occupy different amounts of space.

In the multi-column style option [5], I used page-wide footnotes at the bottom of the page, but again the result doesn't look very pleasant since short footnotes produce undesired gaps of white space. Of course, the main goal of this style option was a balancing algorithm for columns which would allow switching between different numbers of columns on the same page. With this feature, the natural place for footnotes seems to be the bottom of the page[1] but looking at some of the results it seems best to avoid footnotes in such a layout entirely.

Another possibility is to turn footnotes into endnotes, i.e., printing them at the end of every chapter or the end of the entire document. But I assume everyone who has ever read a book using such a layout will agree with me, that it is a pain to search back and forth, so that the reader is tempted to ignore the endnotes entirely.

When I wrote the article about "Future extensions of TeX" [6] I was again dissatisfied with the outcome of the footnotes, and since this article should show certain aspects of high quality typesetting, I decided to give the footnote problem a try and modified the LaTeX output routine for this purpose. The layout I used was inspired by the yearbook of the Gutenberg Gesellschaft Mainz [1]. Later on, I found that it is also recommended by Jan White [9]. On the layout of footnotes I also consulted books by Jan Tschichold [8] and Manfred Simoneit [7], books, I would recommend to everyone being able to read German texts.

1.1 Description of the new layout

The result of this effort is presented in this paper and the reader can judge for himself whether it was successful or not.[2] The main idea for this layout is to assemble the footnotes of all columns on a page and place them all together at the bottom of the right column. Allowing for enough space between footnotes and text, and in addition, setting the footnotes in smaller type[3] I decided that one could omit the footnote separator rule which is used in most publications prepared with TeX.[4] Furthermore, I decided to place the footnote markers[5] at the baseline instead of raising them as superscripts.[6]

All in all, I think this generates a neat layout, and surprisingly enough, the necessary changes to the LaTeX output routine are nevertheless astonishingly simple.

1.2 The use of the style option

This style option might be used together with any other style option for LaTeX which does not change the three internals changed by `ftnright.sty`.[7] In most cases, it is best to use this style option as the very last option in the `\documentstyle` command to make sure that its settings are not overwritten by other options.[8]

[*] The LaTeX style option `ftnright` which is described in this article has the version number v1.0d dated 92/06/19. The documentation was last revised on 92/06/19.

[1] You can not use column footnotes at the bottom, since the number of columns can differ on one page.

[2] Please note, that this option only changed the placement of footnotes. Since this article also makes use of the `doc` option [4], that assigns tiny numbers to code lines sprincled throughout the text, the resulting design is not perfect.

[3] The standard layout in *TUGboat* uses the same size for footnotes and text, giving the footnotes, in my opinion, much too much prominence.

[4] People who prefer the rule can add it by redefining the command `\footnoterule` [2, p. 156]. Please, note, that this command should occupy no space, so that a negative space should be used to compensate for the width of the rule used.

[5] The tiny numbers or symbols, e.g., the '5' in front of this footnote.

[6] Of course, this is only done for the mark preceeding the footnote text and not the one used within the main text where a raised number or symbol set in smaller type will help to keep the flow of thoughts, uninterrupted.

[7] These are the macros `\@startcolumn`, `\@makecol` and `\@outputdblcol` as we will see below. Of course, the option will take only effect with a document style using a twocolumn layout (like `ltugboat`) or when the user additionally specifies `twocolumn` as a document style option in the `\documentstyle` command.

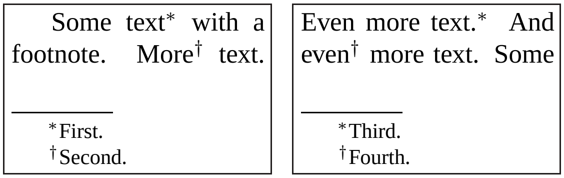
[8] The `ltugboat` option (which is currently set up as a style option instead of a document style option which it actually is) will overwrite

1

Figure 3.2: The placement of text and footnotes with the ftnright package

The interface for footmisc is quite simple: nearly everything is customized by specifying options when the package is loaded, though in some cases further control is possible via parameters.

In the article class, footnotes are numbered sequentially throughout the document; in report and book, footnotes are numbered inside chapters. Sometimes, however, it is more appropriate to number footnotes on a per-page basis. This can be achieved by loading footmisc with the option perpage. The package footnpag (by Joachim Schrod) provides the same feature with a somewhat different implementation as a stand-alone package. A generalized implementation for resetting counters on a per-page basis is provided by the package perpage (see Section 3.2.5 on page 120). Since TEX's page-building mechanism is asynchronous, it is always necessary to process the document at least twice to get the numbering correct. Fortunately, the package warns you via "Rerun to get cross-references right" if the footnote numbers are incorrect. The package stores information between runs in the .aux file, so after a lot of editing this information is sometimes not even close to reality. In such a case deleting the .aux file helps the package to find the correct numbering faster.[1]



Some text* with a footnote. More† text.

*First.
†Second.

Even more text.* And even† more text. Some

*Third.
†Fourth.

```
\usepackage[perpage,symbol]{footmisc}
Some text\footnote{First.} with a footnote.
More\footnote{Second.} text. Even more
text.\footnote{Third.} And even\footnote
 {Fourth.} more text. Some final text.
```

3-2-7

Counter too large errors

For this special occasion our example shows two pages side by side, so you can observe the effects of the perpage option. The example also shows the effect of another option: symbol will use footnote symbols instead of numbers. As only a limited number of such symbols are available, you can use this option only if there are few footnotes in total or if footnote numbers restart on each page. There are six different footnote symbols and, by duplicating some, standard LATEX supports nine footnotes. By triplicating some of them, footmisc supports up to 16 footnotes (per page or in total). If this number is exceeded you will get a LATEX error message.

In particular with the perpage option, this behavior can be a nuisance because the error could be spurious, happening only while the package is still trying to determine which footnotes belong on which page. To avoid this problem, you can use the variant option symbol*, which also produces footnote symbols but numbers footnotes for which there are no symbols left with Arabic numerals. In that case you will get a warning at the end of the run that some footnotes were out of range and detailed information is placed in the transcript file.

```
\setfnsymbol{name}     \DefineFNsymbols*{name}[type]{symbol-list}
```

If the symbol or symbol* option is selected, a default sequence of footnote symbols defined by Leslie Lamport is used. Other authorities suggest different se-

[1] In fact, during the preparation of this chapter I managed to confuse footmisc (by changing the \textheight in an example) so much that it was unable to find the correct numbering thereafter and kept asking for a rerun forever. Removing the .aux file resolved the problem.

lamport	*	†	‡	§	¶	‖	**	††	‡‡	§§	¶¶	***	†††	‡‡‡	§§§	¶¶¶
bringhurst	*	†	‡	§	‖	¶										
chicago	*	†	‡	§	‖	#										
wiley	*	**	†	‡	§	¶	‖									

Table 3.4: Footnote symbol lists predefined by footmisc

quences, so footmisc offers three other sequences to chose from using the declaration \setfnsymbol (see Table 3.4).

In addition, you can define your own sequence using the \DefineFNsymbols declaration in the preamble. It take two mandatory arguments: the *name* to access the list later via \setfnsymbol and the *symbol-list*. From this list symbols are taken one after another (with spaces ignored). If a symbol is built from more than one glyph, it has to be surrounded by braces. If the starred form of the declaration is used, LaTeX issues an error message if it runs out of symbols. Without it, you will get Arabic numerals and a warning at the end of the LaTeX run.

Due to an unfortunate design choice, footnote symbols (as well as some other text symbols) were originally added to the math fonts of TeX, rather than to the text fonts, with the result that they did not change when the text font is modified. In LaTeX this flaw was partly corrected by adding these symbols to the text symbol encoding (TS1; see Section 7.5.4). However, for compatibility reasons the footnote symbols are still taken by default from the math fonts, even though this choice is not appropriate if one has changed the text font from Computer Modern to some other typeface. By using the optional *type* argument with the value text, you can tell footmisc that your list consist of text symbols. Note that all predefined symbol lists consists of math symbols and may need redeclaring if used with fonts other than Computer Modern.

Some text[*] with a footnote. More[**] text. Even more text.[***] And even[****] more text. Some more text to finish up.

[*] First.
[**] Second.
[***] Third.
[****] Fourth.

3-2-8

```
\usepackage[symbol]{footmisc}
\DefineFNsymbols{stars}[text]{* {**} {***} {****}}
\setfnsymbol{stars}
Some text\footnote{First.} with a footnote.
More\footnote{Second.} text. Even more
text.\footnote{Third.} And even\footnote{Fourth.}
more text. Some more text to finish up.
```

If you have many short footnotes then their default placement at the bottom of the page, stacked on top of each other, is perhaps not completely satisfactory. A typical example would be critical editions, which contain many short footnotes.[1] The layout of the footnotes can be changed using the para option, which formats

[1] See, for example, the ledmac package [171] the kinds of footnotes and endnotes that are common in critical editions. This package is a reimplementation of the EDMAC system [112] for LaTeX and was recently made available by Peter Wilson. See also the bigfoot package by David Kastrup.

them into a single paragraph. If this option is chosen then footnotes never split across pages. The code for this option is based on work by Chris Rowley and Dominik Wujastyk (available as the package fnpara), which in turn was inspired by an example in *The TEXbook* by Donald Knuth.

Some text with a footnote.[1] More text.[2] Even more text.[3] Some final text.

[1] A first. [2] A second. [3] A third.

```
\usepackage[para]{footmisc}
Some text with a footnote.\footnote{A first.}
More text.\footnote{A second.}  Even more
text.\footnote{A third.}  Some final text.
```

3-2-9

Another way to deal with footnotes is given by the option side. In this case footnotes are placed into the margin, if possible on the same line where they are referenced. What happens internally is that special \marginpar commands are used to place the footnote text, so everything said in Section 3.2.8 about the \marginpar commands is applicable. This option cannot be used together with the para option, described earlier, but can be combined with most others.

[1]A first.
[2]A second.
[3]A third.
[4]A fourth.

Some text with a footnote.[1] A lot of additional text here with a footnote.[2] Even more text and then another footnote.[3] Some more text.[4] A lot of additional lines of text here to fill up the space on the left.

```
\usepackage[side,flushmargin]{footmisc}
Some text with a footnote.\footnote{A first.}
A lot of additional text here with a
footnote.\footnote{A second.}
Even more text and then another
footnote.\footnote{A third.}
Some more text.\footnote{A fourth.} A lot of
additional lines of text here to fill up the
space on the left.
```

3-2-10

The option flushmargin used in the previous example makes the footnote text start at the left margin with the footnote marker protruding into the margin; by default, the footnote text is indented. For obvious reasons this option is incompatible with the para option. A variant form is called marginal. If this option is used then the marker sticks even farther into the margin, as shown in the example below.

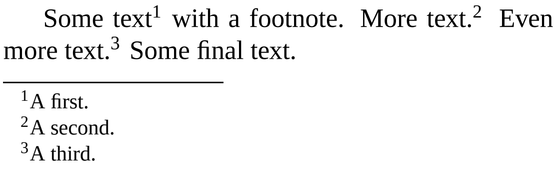
Some text[1] with a footnote. More text.[2] Even more text.[3] Some final text.

[1] A first.
[2] A second.
[3] A third.

```
\usepackage[marginal]{footmisc}
Some text\footnote{A first.} with a
footnote. More text.\footnote{A second.}
Even more text.\footnote{A third.} Some
final text.
```

3-2-11

Instead of using one of the above options, the position of the footnote marker can be directly controlled using the parameter \footnotemargin. If set to a negative value the marker is positioned in the margin. A value of 0pt is equivalent to using the option flushmargin. A positive value means that the footnote text is indented by this amount and the marker is placed flush right in the space produced by the indentation.



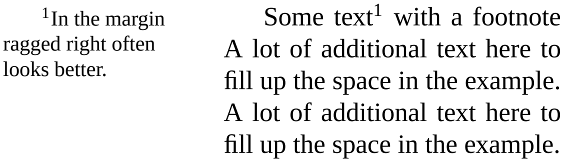
Some text[1] with a footnote. More text.[2] Even more text.[3] Some final text.

[1] A first.
[2] A second.
[3] A third.

3-2-12

```
\usepackage{footmisc}
\setlength\footnotemargin{10pt}
Some text\footnote{A first.} with a
footnote. More text.\footnote{A second.}
Even more text.\footnote{A third.} Some
final text.
```

By default, the footnote text is adjusted but this does not always give satisfactory results, especially with the options `para` and `side`. In case of the `para` option nothing can be done, but for other layouts you can switch to ragged-right typesetting by using the option `ragged`. The next example does not specify `flushmargin`, so we get an indentation of width `\footnotemargin`—compare this to Example 3-2-10 on the preceding page.



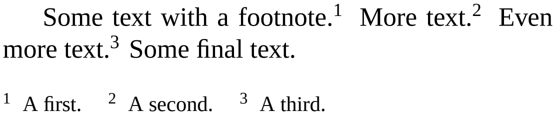
[1] In the margin ragged right often looks better.

Some text[1] with a footnote A lot of additional text here to fill up the space in the example. A lot of additional text here to fill up the space in the example.

3-2-13

```
\usepackage[side,ragged]{footmisc}
Some text\footnote{In the margin ragged
right often looks better.} with a footnote
A lot of additional text here to fill
up the space in the example. A lot of
additional text here to fill up the space
in the example.
```

The two options `norule` and `splitrule` (courtesy of Donald Arseneau) modify the rule normally placed between text and footnotes. If `norule` is specified, then the separation rule will be suppressed. As compensation the value of `\skip\footins` is slightly enlarged. If a footnote does not fit onto the current page it will be split and continued on the next page, unless the `para` option is used (as it does not support split footnotes). By default, the rule separating normal and split footnotes from preceding text is the same. If you specify the option `splitrule`, however, it becomes customizable: the rule above split footnotes will run across the whole column while the one above normal footnotes will retain the default definition given by `\footnoterule`. More precisely, this option will introduce the commands `\mpfootnoterule` (for use in minipages), `\pagefootnoterule` (for use on regular pages), and `\splitfootnoterule` (for use on pages starting with a split footnote). By modifying their definitions, similar to the example given earlier for the `\footnoterule` command, you can customize the layout according to your needs.



Some text with a footnote.[1] More text.[2] Even more text.[3] Some final text.

[1] A first. [2] A second. [3] A third.

3-2-14

```
\usepackage[norule,para]{footmisc}
Some text with a footnote.\footnote{A first.}
More text.\footnote{A second.}  Even more
text.\footnote{A third.}  Some final text.
```

In classes such as article or report in which `\raggedbottom` is in effect, so that columns are allowed to be of different heights, the footnotes are attached at a distance of `\skip\footins` from the column text. If you prefer them aligned at the bottom, so that any excess space is put between the text and the footnotes, specify the option `bottom`. In classes for which `\flushbottom` is in force, such as book, this option does nothing.

In some documents, e.g., literary analysis, several footnotes may appear at a single point. Unfortunately, LaTeX's standard footnote commands are not able to handle this situation correctly: the footnote markers are simply clustered together so that you cannot tell whether you are to look for the footnotes 1 and 2, or for the footnote with the number 12.

Some text[12] with two footnotes. Even more text.[3]

[1] A first. [2] A second. [3] A third.

```
\usepackage[para]{footmisc}
Some text\footnote{A first.}\footnote{A second.} with
two footnotes. Even more text.\footnote{A third.}
```

3-2-15

This problem will be resolved by specifying the option `multiple`, which ensures that footnotes in a sequence will display their markers separated by commas. The separator can be changed to something else, such as a small space, by changing the command `\multfootsep`.

Some text[1,2] with two footnotes. Even more text.[3]

[1] A first. [2] A second. [3] A third.

```
\usepackage[multiple,para]{footmisc}
Some text\footnote{A first.}\footnote{A second.} with
two footnotes. Even more text.\footnote{A third.}
```

3-2-16

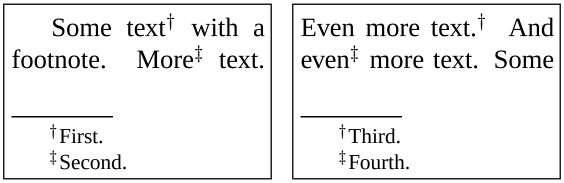
The footmisc package deals with one other potential problem: if you put a footnote into a sectional unit, then it might appear in the table of contents or the running header, causing havoc. Of course, you could prevent this dilemma (manually) by using the optional argument of the heading command; alternatively, you could specify the option `stable`, which prevents footnotes from appearing in such places.

3.2.5 perpage—Resetting counters on a "per-page" basis

As mentioned earlier, the ability to reset arbitrary counters on a per-page basis is implemented in the small package perpage written by David Kastrup.

```
\MakePerPage[start]{counter}
```

The declaration `\MakePerPage` defines *counter* to be reset on every page, optionally requesting that its initial starting value be *start* (default 1). For demonstration we repeat Example 3-2-7 on page 116 but start each footnote marker sequence with the second symbol (i.e., "†" instead of "*").


Some text† with a footnote. More‡ text.

† First.
‡ Second.

Even more text.† And even‡ more text. Some

† Third.
‡ Fourth.

3-2-17

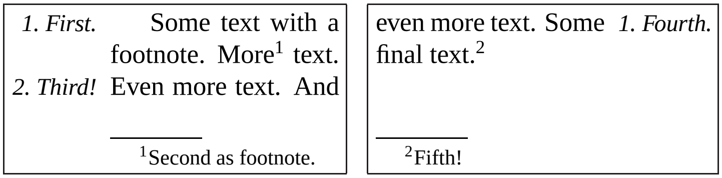
```
\usepackage[symbol]{footmisc}
\usepackage{perpage}
\MakePerPage[2]{footnote}
Some text\footnote{First.} with a footnote.
More\footnote{Second.} text. Even more
text.\footnote{Third.} And even\footnote
 {Fourth.} more text. Some final text.
```

The package synchronizes the numbering via the .aux file of the document, thus requiring at least two runs to get the numbering correct. In addition, you may get spurious "Counter too large" error messages on the first run if \fnsymbol or \alph is used for numbering (see the discussion of the symbol* for the footmisc package on page 116).

Among LaTeX's standard counters probably only footnote can be sensibly modified in this way. Nevertheless, one can easily imagine applications that provide, say, numbered marginal notes, which could be defined as follows:

```
\newcounter{mnote}
\newcommand\mnote[1]{{\refstepcounter{mnote}%
  \marginpar[\itshape\small\raggedleft\themnote.\ #1]%
            {\itshape\small\raggedright\themnote.\ #1}}}
\usepackage{perpage} \MakePerPage{mnote}
```

We step the new counter mnote outside the \marginpar so that it is executed only once;[1] we also need to limit the scope of the current redefinition of \label (through \refstepcounter) so we put braces around the whole definition. Notes on left-hand pages should be right aligned, so we use the optional argument of \marginpar to provide different formatting for this case.


1. First. Some text with a footnote. More[1] text.
2. Third! Even more text. And

[1] Second as footnote.

even more text. Some final text.[2] *1. Fourth.*

[2] Fifth!

3-2-18

```
% code as above
Some text\mnote{First.} with a
footnote. More\footnote{Second
as footnote.} text. Even more
text.\mnote{Third!} And even
more\mnote {Fourth.} text. Some
final text.\footnote{Fifth!}
```

Another application for the package is given in Example 3-2-24 on page 125, where several independent footnote streams are all numbered on a per-page basis.

[1] If placed in both arguments of \marginpar it would be executed twice. It would work if placed in the optional argument only, but then we would make use of an implementation detail (that the optional argument is evaluated first) that may change.

3.2.6 manyfoot—Independent footnotes

Most documents have only a few footnotes, if any. For them LaTeX's standard commands plus the enhancements offered by footmisc are usually sufficient. However, certain applications, such as critical editions, require several independently numbered footnote streams. For these situations the package manyfoot by Alexander Rozhenko can provide valuable help.[1]

`\DeclareNewFootnote[`*fn-style*`]{`*suffix*`}[`*enum-style*`]`

This declaration can be used to introduce a new footnote level. In its simplest form you merely specify a *suffix* such as "`B`". This allocates a new counter `footnote`⟨*suffix*⟩ that is used to automatically number the footnotes on the new level. The default is to use Arabic numerals; by providing the optional argument *enum-style*, some other counter style (e.g., `roman` or `alph`) can be selected.

The optional *fn-style* argument defines the general footnote style for the new level; the default is `plain`. If the package was loaded with the `para` or `para*` option, then `para` can also be selected as the footnote style.

The declaration will then automatically define six commands for you. The first three are described here:

`\footnote`⟨*suffix*⟩`[`*number*`]{`*text*`}` Same as `\footnote` but for the new level. Steps the `footnote`⟨*suffix*⟩ counter unless the optional *number* argument is given. Generates footnote markers and puts *text* at the bottom of the page.

`\footnotemark`⟨*suffix*⟩`[`*number*`]` Same as `\footnotemark` but for the new level. Steps the corresponding counter (if no optional argument is used) and prints a footnote marker corresponding to its value.

`\footnotetext`⟨*suffix*⟩`[`*number*`]{`*text*`}` Same as `\footnotetext` but for the new level. Puts *text* at the bottom of the page using the current value of `footnote`⟨*suffix*⟩ or the optional argument to generate a footnote marker in front of it.

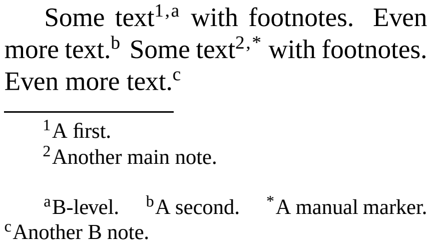
In all three cases the style of the markers depends on the chosen *enum-style*.

The remaining three commands defined by `\DeclareNewFootnote` for use in the document are `\Footnote`⟨*suffix*⟩, `\Footnotemark`⟨*suffix*⟩, and `\Footnotetext`⟨*suffix*⟩ (i.e., same names as above but starting with an uppercase `F`). The important difference to the previous set is the following: instead of the optional *number* argument, they require a mandatory *marker* argument allowing you to specify arbitrary markers if desired. Some examples are given below.

The layout of the footnotes can be influenced by loading the footmisc package in addition to manyfoot, except that the `para` option of footmisc cannot be used. In the next example we use the standard footnote layout for top-level footnotes and the run-in layout (option `para`) for the second level. Thus, if all footnote levels should produce run-in footnotes, the solution is to avoid top-level footnotes

[1] A more comprehensive package, bigfoot, is currently being developed by David Kastrup.

completely (e.g., \footnote) and provide all necessary levels through manyfoot. Note how footmisc's multiple option properly acts on all footnotes.



Some text[1,a] with footnotes. Even more text.[b] Some text[2,*] with footnotes. Even more text.[c]

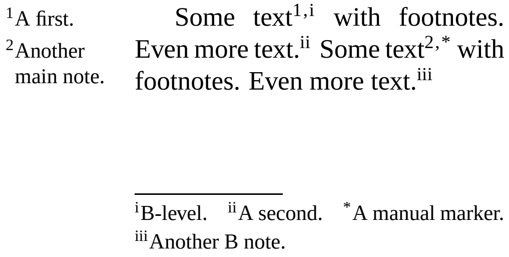
[1] A first.
[2] Another main note.

[a] B-level. [b] A second. [*] A manual marker. [c] Another B note.

```
\usepackage[multiple]{footmisc}
\usepackage[para]{manyfoot}
\DeclareNewFootnote[para]{B}[alph]
Some text\footnote{A first.}\footnoteB{B-level.}
with footnotes. Even more text.\footnoteB{A second.}
Some text\footnote{Another main note.}%
\FootnoteB{*}{A manual marker.} with footnotes.
Even more text.\footnoteB{Another B note.}
```

3-2-19

In the following example the top-level footnotes are moved into the margin by loading footmisc with a different set of options. This time manyfoot is loaded with the option para*, which differs from the para option used previously in that it suppresses any indentation for the run-in footnote block. In addition, the second-level notes are now numbered with Roman numerals. For comparison the example typesets the same input text as Example 3-2-19 but it uses a different measure, as we have to show marginal notes now.



[1] A first.
[2] Another main note.

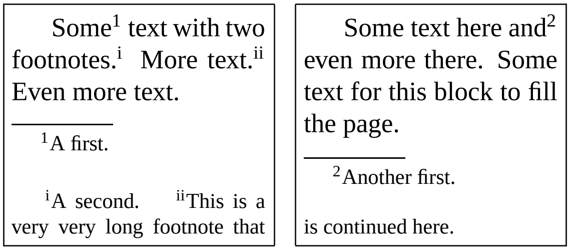
Some text[1,i] with footnotes. Even more text.[ii] Some text[2,*] with footnotes. Even more text.[iii]

[i] B-level. [ii] A second. [*] A manual marker. [iii] Another B note.

```
\usepackage[side,flushmargin,ragged,multiple]
           {footmisc}
\usepackage[para*]{manyfoot}
\DeclareNewFootnote[para]{B}[roman]
Some text\footnote{A first.}\footnoteB{B-level.}
with footnotes. Even more text.\footnoteB{A
second.} Some text\footnote{Another main note.}%
\FootnoteB{*}{A manual marker.} with footnotes.
Even more text.\footnoteB{Another B note.}
```

3-2-20

The use of run-in footnotes, with either the para or the para* option, is likely to produce one particular problem: very long footnotes near a page break will not be split. To resolve this problem the manyfoot package offers a (semi)manual solution: at the point where you wish to split your note you place a \SplitNote command and end the footnote. You then place the remaining text of the footnote one paragraph farther down in the document in a \Footnotetext⟨*suffix*⟩ using an empty *marker* argument.



Some[1] text with two footnotes.[i] More text.[ii] Even more text.

[1] A first.

[i] A second. [ii] This is a very very long footnote that

Some text here and[2] even more there. Some text for this block to fill the page.

[2] Another first.

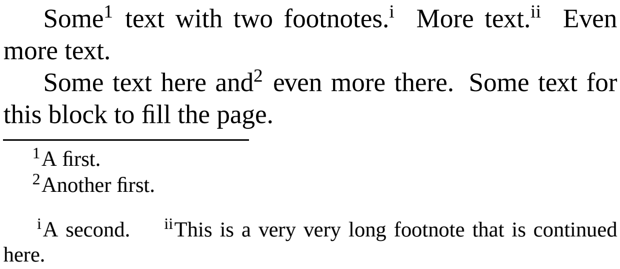
is continued here.

```
\usepackage[para]{manyfoot}
\DeclareNewFootnote[para]{B}[roman]
Some\footnote{A first.} text with two
footnotes.\footnoteB{A second.} More
text.\footnoteB{This is a very very long
footnote that\SplitNote} Even more text.

Some\FootnotetextB{}{is continued here.}
text here and\footnote{Another first.}
even more there. \sample % as elsewhere
```

3-2-21

If both parts of the footnote fall onto the same page after reformatting the document, the footnote parts get correctly reassembled, as we prove in the next example, which uses the same example text but a different measure. However, if the reformatting requires breaking the footnote in a different place, then further manual intervention is unavoidable. Thus, such work is best left until the last stage of production.



Some[1] text with two footnotes.[i] More text.[ii] Even more text.

Some text here and[2] even more there. Some text for this block to fill the page.

[1] A first.

[2] Another first.

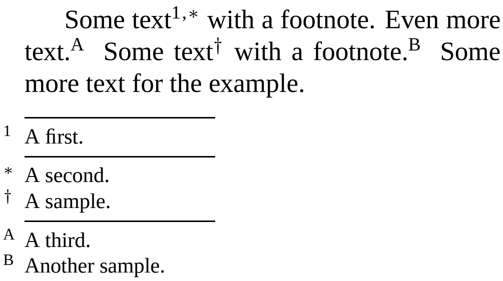
[i] A second. [ii] This is a very very long footnote that is continued here.

```
\usepackage[para]{manyfoot}
\DeclareNewFootnote[para]{B}[roman]
Some\footnote{A first.} text with two
footnotes.\footnoteB{A second.} More
text.\footnoteB{This is a very very long
footnote that\SplitNote} Even more text.

Some\FootnotetextB{}{is continued here.}
text here and\footnote{Another first.}
even more there. \sample % as elsewhere
```

3-2-22

The vertical separation between a footnote block and the previous one is specified by `\skip\footins`⟨*suffix*⟩. By default, it is equal to `\skip\footins` (i.e., the separation between main text and footnotes). Initially the extra blocks are only separated by such spaces, but if the option `ruled` is included a `\footnoterule` is used as well. In fact, arbitrary material can be placed in that position by redefining the command `\extrafootnoterule`—the only requirement being that the typeset result from that command does not take up any additional vertical space (see the discussion of `\footnoterule` on page 112 for further details). It is even possible use different rules for different blocks of footnotes; consult the package documentation for details.



Some text[1,*] with a footnote. Even more text.[A] Some text[†] with a footnote.[B] Some more text for the example.

[1] A first.

[*] A second.

[†] A sample.

[A] A third.

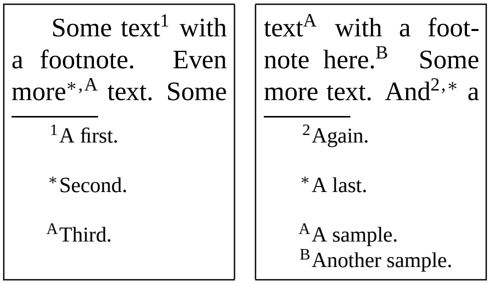
[B] Another sample.

```
\usepackage[marginal,multiple]{footmisc}
\usepackage[ruled]{manyfoot}
\DeclareNewFootnote{B}[fnsymbol]
\DeclareNewFootnote{C}[Alph]
\setlength{\skip\footinsB}{5pt minus 1pt}
\setlength{\skip\footinsC}{5pt minus 1pt}
Some text\footnote{A first.}\footnoteB{A second.}
with a footnote. Even more text.\footnoteC{A third.}
Some text\footnoteB{A sample.} with a
footnote.\footnoteC{Another sample.} Some more
text for the example.
```

3-2-23

Number the footnotes per page

The previous example deployed two additional *enum-style*s, `Alph` and `fnsymbol`. However, as only a few footnote symbols are available in both styles, that choice is most likely not a good one, unless we ensure that these footnote streams are numbered on a per-page basis. The `perpage` option of `footmisc` will not help here, as it applies to only the top-level footnotes. We can achieve the desired effect either by using \MakePerPage from the perpage package on the counters footnoteB and footnoteC (as done below), or by using the perpage option of manyfoot (which calls on the perpage package to do the job, which will number all new footnote levels defined on a per-page basis). Note that the top-level footnotes are still numbered sequentially the way the example was set up.



Some text[1] with a footnote. Even more[*,A] text. Some

[1] A first.

[*] Second.

[A] Third.

text[A] with a footnote here.[B] Some more text. And[2,*] a

[2] Again.

[*] A last.

[A] A sample.
[B] Another sample.

3-2-24

```
\usepackage[multiple]{footmisc}
\usepackage{manyfoot,perpage}
\DeclareNewFootnote{B}[fnsymbol]
\DeclareNewFootnote{C}[Alph]
\MakePerPage{footnoteB}\MakePerPage{footnoteC}
Some text\footnote{A first.} with a footnote.
Even more\footnoteB{Second.}\footnoteC{Third.}
text. Some text\footnoteC{A sample.} with a
footnote here.\footnoteC{Another sample.} Some
more text. And\footnote{Again.}\footnoteB{A
  last.} a last note.
```

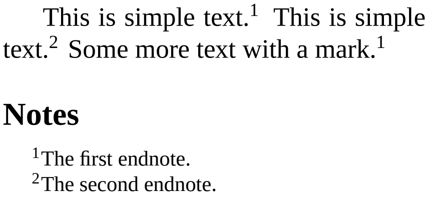
3.2.7 endnotes—An alternative to footnotes

Scholarly works usually group notes at the end of each chapter or at the end of the document. Such notes are called endnotes. Endnotes are not supported in standard LaTeX, but they can be created in several ways.

The package endnotes (by John Lavagnino) provides its own \endnote command, thus allowing footnotes and endnotes to coexist.

The document-level syntax is modeled after the footnote commands if you replace foot with end—for example, \endnote produces an endnote, \endnotemark produces just the mark, and \endnotetext produces just the text. The counter used to hold the current endnote number is called endnote and is stepped whenever \endnote or \endnotemark without an optional argument is used.

All endnotes are stored in an external file with the extension .ent and are made available when you issue the command \theendnotes.



This is simple text.[1] This is simple text.[2] Some more text with a mark.[1]

Notes

[1] The first endnote.
[2] The second endnote.

3-2-25

```
\usepackage{endnotes}
This is simple text.\endnote{The first endnote.}
This is simple text.\endnote{The second endnote.}
Some more text with a mark.\endnotemark[1]

\theendnotes   % output endnotes here
```

This process is different from the way the table of contents is built; the endnotes are written directly to the file, so that you will see only those endnotes which are defined earlier in the document. The advantage of this approach is that you can have several calls to \theendnotes, for example, at the end of each chapter.

To additionally restart the numbering you have to set the `endnote` counter to zero after calling `\theendnotes`.

The heading produced by `\theendnotes` can be controlled in several ways. The text can be changed by modifying `\notesname` (default is the string `Notes`). If that is not enough you can redefine `\enoteheading`, which is supposed to produce the sectioning command in front of the notes.

The layout for endnote numbers is controlled through `\theendnote`, which is the standard way LaTeX handles counter formatting. The format of the mark is produced from `\makeenmark` with `\theenmark`, holding the formatted number for the current mark.

This is simple text.[a)] This is simple text.[b)] Some more text with a mark.[a)]

Chapter Notes

[a)] The first endnote.
[b)] The second endnote.

```
\usepackage{endnotes}
\renewcommand\theendnote{\alph{endnote}}
\renewcommand\makeenmark{\textsuperscript{\theenmark)}}
\renewcommand\notesname {Chapter Notes}
This is simple text.\endnote{The first endnote.}
This is simple text.\endnote{The second endnote.}
Some more text with a mark.\endnotemark[1]
\theendnotes
```

3-2-26

The font size for the list of endnotes is controlled through `\enotesize`, which defaults to `\footnotesize`. Also, by modifying `\enoteformat` you can change the display of the individual endnotes within their list. This command is supposed to set up the paragraph parameters for the endnotes and to typeset the note number stored in `\theenmark`. In the example we start with no indentation for the first paragraph and with the number placed into the margin.

This is simple text.[1] This is simple text.[2] Some more text with a mark.[1]

Notes

1. The first endnote with a lot of text to produce two lines.
   And even a second paragraph.
2. The second endnote.

```
\usepackage{endnotes}
\renewcommand\enoteformat{\noindent\raggedright
 \setlength\parindent{12pt}\makebox[0pt][r]{\theenmark.\,}}
\renewcommand\enotesize{\scriptsize}
This is simple text.\endnote{The first endnote with a lot
   of text to produce two lines.\par And even a second
   paragraph.}
This is simple text.\endnote{The second endnote.}
Some more text with a mark.\endnotemark[1]
\theendnotes
```

3-2-27

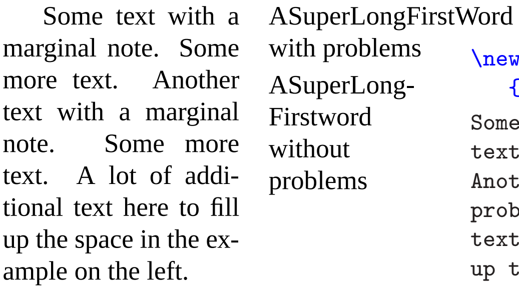
3.2.8 Marginal notes

The standard LaTeX command `\marginpar` generates a marginal note. This command typesets the text given as its argument in the margin, with the first line being at the same height as the line in the main text where the `\marginpar` command occurs. When only the mandatory argument is specified, the text goes to the right margin for one-sided printing; to the outside margin for two-sided printing;

and to the nearest margin for two-column formatting. When you also specify an optional argument, its text is used if the left margin is chosen, while the second (mandatory) argument is used for the right margin.

This placement strategy can be reversed (except for two-column formatting) using `\reversemarginpar`, which acts on all marginal notes from there on. You can return to the default behavior with `\normalmarginpar`.

There are a few important things to understand when using marginal notes. First, the `\marginpar` command does not start a paragraph. Thus, if it is used before the first word of a paragraph, the vertical alignment will not match the beginning of the paragraph. Second, the first word of its argument is not automatically hyphenated. Thus, for a narrow margin and long words (as in German), you may have to precede the first word by a `\hspace{0pt}` command to allow hyphenation of that word. These two potential problems can be eased by defining a command like `\marginlabel`, which starts with an empty box `\mbox{}`, typesets a marginal note ragged left, and adds a `\hspace{0pt}` in front of the argument.



Some text with a marginal note. Some more text. Another text with a marginal note. Some more text. A lot of additional text here to fill up the space in the example on the left.

ASuperLongFirstWord with problems

ASuperLongFirstword without problems

3-2-28

```
\newcommand\marginlabel[1]{\mbox{}\marginpar
   {\raggedright\hspace{0pt}#1}}
Some\marginpar{ASuperLongFirstWord with problems}
text with a marginal note. Some more text.
Another\marginlabel{ASuperLongFirstword without
problems}  text with a marginal note. Some more
text. A lot of additional text here to fill
up the space in the example on the left.
```

Of course, the above definition can no longer produce different texts depending on the chosen margin. With a little more finesse this problem could be solved, using, for example, the `\ifthenelse` constructs from the ifthen package.

Incorrectly placed `\marginpar`*s*

The LaTeX kernel tries hard (without producing too much processing overhead) to ensure that the contents of `\marginpar` commands always show up in the correct margin and in most circumstances will make the right decisions. In some cases, however, it will fail. If you are unlucky enough to stumble across one of them, a one-off solution is to add an explicit `\pagebreak` to stop the page generation from looking too far ahead. Of course, this has the disadvantage that the correction means visual formatting and has to be undone if the document changes. A better solution is to load the package mparhack written by Tom Sgouros and Stefan Ulrich. Once this package is loaded all `\marginpar` positions are tracked (internally using a label mechanism and writing the information to the `.aux` file). You may then get a warning "Marginpars may have changed. Rerun to get them right", indicating that the positions have changed in comparison to the previous LaTeX run and that a further run is necessary to stabilize the document.

As explained in Table 4.2 on page 196, there are three length parameters to customize the style of marginal notes: `\marginparwidth`, `\marginparsep`, and `\marginparpush`.

	Command	*Default Definition*	*Representation*
First Level	`\labelitemi`	`\textbullet`	•
Second Level	`\labelitemii`	`\normalfont\bfseries \textendash`	–
Third Level	`\labelitemiii`	`\textasteriskcentered`	∗
Fourth Level	`\labelitemiv`	`\textperiodcentered`	·

Table 3.5: Commands controlling an `itemize` list environment

3.3 List structures

Lists are very important LaTeX constructs and are used to build many of LaTeX's display-like environments. LaTeX's three standard list environments are discussed in Section 3.3.1, where we also show how they can be customized. Section 3.3.2 starting on page 132 provides an in-depth discussion of the paralist package, which introduces a number of new list structures and offers comprehensive methods to customize them, as well as the standard lists. It is followed by a discussion of "headed lists", such as theorems and exercises. Finally, Section 3.3.4 on page 144 discusses LaTeX's general list environment.

3.3.1 Modifying the standard lists

It is relatively easy to customize the three standard LaTeX list environments `itemize`, `enumerate`, and `description`, and the next three sections will look at each of these environments in turn. Changes to the default definitions of these environments can either be made globally by redefining certain list-defining parameters in the document preamble or can be kept local.

Customizing the `itemize` list environment

For a simple unnumbered `itemize` list, the labels are defined by the commands shown in Table 3.5. To create a list with different-looking labels, you can redefine the label-generating command(s). You can make that change local for one list, as in the example below, or you can make it global by putting the redefinition in the document preamble. The following simple list is a standard `itemize` list with a marker from the PostScript Zapf Dingbats font (see Section 7.6.4 on page 378) for the first-level label:

```
\usepackage{pifont}
\newenvironment{MYitemize}{\renewcommand\labelitemi
          {\ding{43}}\begin{itemize}}{\end{itemize}}
```

☞ Text of the first item in the list.

☞ Text of the first sentence in the second item of the list. And the second sentence.

```
\begin{MYitemize}
\item Text of the first item in the list.
\item Text of the first sentence in the second
      item of the list. And the second sentence.
\end{MYitemize}
```

3-3-1

Customizing the `enumerate` list environment

LaTeX's enumerated (numbered) list environment `enumerate` is characterized by the commands and representation forms shown in Table 3.6 on the next page. The first row shows the names of the counter used for numbering the four possible levels of the list. The second and third rows are the commands giving the representation of the counters and their default definition in the standard LaTeX class files. Rows four, five, and six contain the commands, the default definition, and an example of the actual enumeration string printed by the list.

A reference to a numbered list element is constructed using the `\theenumi`, `\theenumii`, and similar commands, prefixed by the commands `\p@enumi`, `\p@enumii`, etc., respectively. The last three rows in Table 3.6 on the following page show these commands, their default definition, and an example of the representation of such references. It is important to consider the definitions of both the representation and reference-building commands to get the references correct.

We can now create several kinds of numbered description lists simply by applying what we have just learned.

Our first example redefines the first- and second-level counters to use capital Roman digits and Latin characters. The visual representation should be the value of the counter followed by a dot, so we can use the default value from Table 3.6 on the next page for `\labelenumi`.

I. **Introduction**

  A. **Applications**
  Motivation for research and applications related to the subject.

  B. **Organization**
  Explain organization of the report, what is included, and what is not.

II. **Literature Survey**

3-3-2 q1=I q2=IA q3=IB q4=II

```
\renewcommand\theenumi   {\Roman{enumi}}
\renewcommand\theenumii  {\Alph{enumii}}
\renewcommand\labelenumii{\theenumii.}
\begin{enumerate}
  \item \textbf{Introduction}        \label{q1}
  \begin{enumerate}
    \item \textbf{Applications}  \\
      Motivation for research and applications
      related to the subject.        \label{q2}
    \item \textbf{Organization}  \\
      Explain organization of the report, what
      is included, and what is not.  \label{q3}
  \end{enumerate}
  \item \textbf{Literature Survey}   \label{q4}
\end{enumerate}
q1=\ref{q1} q2=\ref{q2} q3=\ref{q3} q4=\ref{q4}
```

After these redefinitions we get funny-looking references; to correct this we have to adjust the definition of the prefix command `\p@enumii`. For example, to get a reference like "I–A" instead of "IA" as in the previous example, we need

```
\makeatletter \renewcommand\p@enumii{\theenumi--} \makeatother
```

because the reference is typeset by executing `\p@enumii` followed by `\theenumii`.

	First Level	*Second Level*	*Third Level*	*Fourth Level*
Counter	`enumi`	`enumii`	`enumiii`	`enumiv`
Representation	`\theenumi`	`\theenumii`	`\theenumiii`	`\theenumiv`
Default Definition	`\arabic{enumi}`	`\alph{enumii}`	`\roman{enumiii}`	`\Alph{enumiv}`
Label Field	`\labelenumi`	`\labelenumii`	`\labelenumiii`	`\labelenumiv`
Default Form	`\theenumi.`	`(\theenumii)`	`\theenumiii.`	`\theenumiv.`
Numbering Example	1., 2.	(a), (b)	i., ii.	A., B.
		Reference representation		
Prefix	`\p@enumi`	`\p@enumii`	`\p@enumiii`	`\p@enumiv`
Default Definition	`{}`	`\theenumi`	`\theenumi(\theenumii)`	`\p@enumiii\theenumiii`
Reference Example	1, 2	1a, 2b	1(a)i, 2(b)ii	1(a)iA, 2(b)iiB

Table 3.6: Commands controlling an `enumerate` list environment

Note that we need `\makeatletter` and `\makeatother` because the command name to redefine contains an `@` sign. Instead of this low-level method, consider using `\labelformat` from the varioref package described in Section 2.4.2.

You can also decorate an `enumerate` field by adding something to the label field. In the example below, we have chosen for the first-level list elements the paragraph sign (§) as a prefix and a period as a suffix (omitted in references).

§1. text inside list, more text inside list

§2. text inside list, more text inside list

§3. text inside list, more text inside list

w1=§1 w2=§2 w3=§3

```
\renewcommand\labelenumi{\S\theenumi.}
\usepackage{varioref} \labelformat{enumi}{\S#1}
\begin{enumerate}
\item \label{w1} text inside list, more text inside list
\item \label{w2} text inside list, more text inside list
\item \label{w3} text inside list, more text inside list
\end{enumerate}
w1=\ref{w1}  w2=\ref{w2}   w3=\ref{w3}
```

3-3-3

You might even want to select different markers for consecutive labels. For instance, in the following example, characters from the PostScript font ZapfDingbats are used. In this case there is no straightforward way to automatically make the `\ref` commands produce the correct references. Instead of `\theenumi` simply producing the representation of the `enumi` counter, we define it to calculate from the counter value which symbol to select. The difficulty here is to create this definition in a way such that it survives the label-generating process. The trick is to add the `\protect` commands so that `\setcounter` and `\ding` are not executed when the label is written to the `.aux` file, yet to ensure that the current value of the counter is stored therein. The latter goal is achieved by prefixing `\value` by the (internal)

TEX command `\the` within `\setcounter` (but not within `\ding`!); without it the references would all show the same values.[1]

① text inside list, text inside list, text inside list, more text inside list;

② text inside list, text inside list, text inside list, more text inside list;

③ text inside list, text inside list, text inside list, more text inside list.

3-3-4 l1=① l2=② l3=③

```
\usepackage{calc,pifont}  \newcounter{local}
\renewcommand\theenumi{\protect\setcounter{local}%
  {171+\the\value{enumi}}\protect\ding{\value{local}}}
\renewcommand\labelenumi{\theenumi}
\begin{enumerate}
\item text inside list, text inside list, \label{l1}
      text inside list, more text inside list;
\item text inside list, text inside list, \label{l2}
      text inside list, more text inside list;
\item text inside list, text inside list, \label{l3}
      text inside list, more text inside list.
\end{enumerate}
l1=\ref{l1}  l2=\ref{l2}  l3=\ref{l3}
```

The same effect is obtained with the `dingautolist` environment defined in the pifont package, which is part of the PSNFSS system (see Section 7.6.4 on page 378).

Customizing the `description` list environment

With the `description` environment you can change the `\descriptionlabel` command that generates the label. In the following example the font for typesetting the labels is changed from boldface (default) to sans serif.

A. text inside list, text inside list, text inside list, more text inside list;

B. text inside list, text inside list, text inside list, more text inside list;

3-3-5

```
\renewcommand\descriptionlabel[1]%
         {\hspace{\labelsep}\textsf{#1}}
\begin{description}
\item[A.] text inside list,  text inside list,
      text inside list, more text inside list;
\item[B.] text inside list,  text inside list,
      text inside list, more text inside list;
\end{description}
```

The standard LATEX class files set the starting point of the label box in a `description` environment at a distance of `\labelsep` to the left of the left margin of the enclosing environment. Thus, the `\descriptionlabel` command in the example above first adds a value of `\labelsep` to start the label aligned with the left margin (see page 147 for detailed explanations).

[1] For the TEXnically interested: LATEX's `\value` command, despite its name, is not producing the "value" of a LATEX counter but only its internal TEX register name. In most circumstances this can be used as the value but unfortunately not inside `\edef` or `\write`, where the internal name rather than the "value" will survive. By prefixing the internal register name with the command `\the`, we get the "value" even in such situations.

3.3.2 paralist—Extended list environments

The paralist package created by Bernd Schandl provides a number of new list environments and offers extensions to LaTeX's standard ones that make their customization much easier. Standard and new list environments can be nested within each other and the enumeration environments support the `\label`/`\ref` mechanism.

Enumerations

All standard LaTeX lists are display lists; that is, they leave some space at their top and bottom as well as between each item. Sometimes, however, one wishes to enumerate something within a paragraph without such visual interruption. The `inparaenum` environment was developed for this purpose. It supports an optional argument that you can use to customize the generated labels, the exact syntax of which is discussed later in this section.

We may want to enumerate items within a paragraph to (a) save space (b) make a less prominent statement, or (c) for some other reason.

```
\usepackage{paralist}
We may want to enumerate items within a paragraph to
\begin{inparaenum}[(a)]
  \item save space
  \item make a less prominent statement, or
  \item for some other reason.
\end{inparaenum}
```

3-3-6

But perhaps this is not precisely what you are looking for. A lot of people like to have display lists but prefer them without much white space surrounding them. In that case `compactenum` might be your choice, as it typesets the list like `enumerate` but with all vertical spaces set to `0pt`.

On the other hand we may want to enumerate like this:

i) still make a display list
ii) format items as usual but with less vertical space, that is
iii) similar to normal `enumerate`.

```
\usepackage{paralist}
On the other hand we may want to enumerate like this:
\begin{compactenum}[i)]
 \item still make a display list
 \item format items  as usual but with less
       vertical space, that is
 \item similar to normal \texttt{enumerate}.
\end{compactenum}
```

3-3-7

Actually, our previous statement was not true—you can customize the vertical spaces used by `compactenum`. Here are the parameters: `\pltopsep` is the space above and below the environment, `\plpartopsep` is the extra space added to the previous space when the environment starts a paragraph on its own, `\plitemsep` is the space between items, and `\plparsep` is the space between paragraphs within an item.

A final enumeration alternative is offered with the asparaenum environment, which formats the items as individual paragraphs. That is, their first line is indented by \parindent and following lines are aligned with the left margin.

Or perhaps we may want to enumerate like this:

1) still make a display list

2) format items as paragraphs with turnover lines not indented, that is

3) similar to normal enumerate.

3-3-8

```
\usepackage{paralist}
Or perhaps we may want to enumerate like this:
\begin{asparaenum}[1)]
\item still make a display list \item format items
 as paragraphs with turnover lines not indented,
 that is \item similar to normal \texttt{enumerate}.
\end{asparaenum}
```

As seen in the previous examples all enumeration environments support one optional argument that describes how to format the item labels. Within the argument the tokens A, a, I, i, and 1 have a special meaning: they are replaced by the enumeration counter displayed in style \Alph, \alph, \Roman, \roman, or \arabic, respectively. All other characters retain their normal meanings. Thus, the argument [(a)] will result in labels like (a), (b), (c), and so on, while [\S i:] will produce §i:, §ii:, §iii:, and so on.

You have to be a bit careful if your label contains text strings, such as labels like Example 1, Example 2, ... In this case you have to hide the "a" inside a brace group—that is, use an argument like [{Example} 1]. Otherwise, you will get strange results, as shown in the next example.

Item b shows what can go wrong:

Example a: On the first item we will not notice it but

Exbmple b: the second item then shows what happens if a special character is mistakenly matched.

3-3-9

```
\usepackage{paralist}
Item~\ref{bad} shows what can go wrong:
\begin{asparaenum}[Example a:]
\item On the first item we will not notice it
but \item the second item then shows what
happens if a special character is mistakenly
matched. \label{bad}
\end{asparaenum}
```

Fortunately, the package usually detects such incorrect input and will issue a warning message. A consequence of hiding special characters by surrounding them with braces is that an argument like [\textbf{a)}] will not work either, because the "a" will not be considered as special any more. A workaround for this case is to use something that does not require braces, such as \bfseries.

As can be seen above, referencing a \label will produce only the counter value in the chosen representation but not any frills added in the optional argument. This is the case for all enumeration environments.

It is not possible with this syntax to specify that a label should show the outer as well as the inner enumeration counter, because the special characters always refer to the current enumeration counter. There is one exception: if you load the package with the option pointedenum or with the option pointlessenum, you will get labels like those shown in the next example.

```
\usepackage[pointedenum]{paralist}
\begin{compactenum}
\item First level.
 \begin{compactenum}
 \item Second level.
  \begin{compactenum} \item Third level. \end{compactenum}
 \item Second level again.
 \end{compactenum}
\end{compactenum}
```

1. First level.
   1.1. Second level.
      1.1.1. Third level.
   1.2. Second level again.

3-3-10

The difference between the two options is the presence or absence of the trailing period. As an alternative to the options you can use the commands \pointedenum and \pointlessenum. They enable you to define your own environments that format labels in this way while other list environments show labels in different formats. If you need more complicated labels, such as those involving several enumeration counters from different levels, then you have to construct them manually using the methods described in Section 3.3.1 on page 129.

The optional argument syntax for specifying the typesetting of enumeration labels was first implemented in the enumerate package by David Carlisle, who extended the standard enumerate environment to support such an optional argument. With paralist the optional argument is supported for all enumeration environments, including the standard enumerate environment (for which it is an upward-compatible extension).

If an optional argument is used on any of the enumeration environments, then by default the left margin will be made only as wide as necessary to hold the labels. More exactly, the indentation is adjusted to the width of the label as it would be if the counter value is currently seven. This produces a fairly wide number (vii) if the numbering style is “Roman” and does not matter otherwise. This behavior is shown in the next example. For some documents this might be the right behavior, but if you prefer a more uniform indentation use the option neverdecrease, which will ensure that the left margin is always at least as wide as the default setting.

```
\usepackage{paralist}
The left margin may vary if we are not careful.
\begin{enumerate}
\item An item in a normal \texttt{enumerate}.
\end{enumerate}
\begin{compactenum}
\item Same left margin in \item this case.
\end{compactenum}
\begin{compactenum}[i)]
\item But a different one \item here.
\end{compactenum}
```

The left margin may vary if we are not careful.

1. An item in a normal enumerate.

1. Same left margin in
2. this case.

i) But a different one
ii) here.

3-3-11

On the other hand, you can always force that kind of adjustment, even for environments without an optional argument, by specifying the option `alwaysadjust`.

Here we force the shortest possible indentation always:

1. An item in a normal `enumerate`.

i) But a different
ii) indentation
iii) here.
1. Same left margin as
2. in the first case.

3-3-12

```
\usepackage[alwaysadjust]{paralist}
Here we force the shortest possible indentation always:
\begin{enumerate}
\item An item in a normal \texttt{enumerate}.
\end{enumerate}
\begin{compactenum}[i)]
\item But a different \item indentation \item here.
\end{compactenum}
\begin{compactenum}[1.]
\item Same left margin as \item in the first case.
\end{compactenum}
```

Finally, with the option `neveradjust` the standard indentation is used in all cases. Thus, labels that are too wide will extend into the left margin.

With this option the label is pushed into the margin.

1. An item in a normal
`enumerate`.

Task A) Same left indentation in
Task B) this case.
1) And the same indentation
2) here.

3-3-13

```
\usepackage[neveradjust]{paralist}
With this option the label is pushed into the margin.
\begin{enumerate}
\item An item in a normal\\ \texttt{enumerate}.
\end{enumerate}
\begin{compactenum}[{Task} A)]
\item Same left indentation in \item this case.
\end{compactenum}
\begin{compactenum}[1)]
\item And the same indentation \item here.
\end{compactenum}
```

Itemizations

For itemized lists the `paralist` package offers the environments `compactitem`, which is a compact version of the standard `itemize` environment; `asparaitem` which formats the items as paragraphs; and `inparaitem`, which produces an inline itemization. The last environment was added mainly for symmetry reasons. All three environments accept an optional argument, that specifies the label to be used for each item.

Producing itemized lists with special labels is easy.

⋆ This example uses the package option `neverdecrease`.
⋆ Without it the left margin would be smaller.

3-3-14

```
\usepackage[neverdecrease]{paralist}
Producing itemized lists with special labels is easy.
\begin{compactitem}[$\star$]
\item This example uses the package option
      \texttt{neverdecrease}.
\item Without it the left margin would be smaller.
\end{compactitem}
```

The three label justification options `neverdecrease`, `alwaysadjust`, and `neveradjust` are also valid for the itemized lists, as can be seen in the previous example. When the paralist package is loaded, LaTeX's `itemize` environment is extended to also support that type of optional argument.

Descriptions

For descriptions the paralist package introduces three additional environments: `compactdesc`, which is like the standard LaTeX `description` environment but with all vertical spaces reduced to zero (or whatever you specify as a customization); `asparadesc`, which formats each item as a paragraph; and `inparadesc`, which allows description lists within running text.

Because description-type environments specify each label at the `\item` command, these environments have no need for an optional argument.

Do you like inline description lists? Try them out!

paralist A useful package as it supports **compact...** environments that have zero vertical space, **aspara...** environments formatted as paragraphs, and **inpara...** environments as inline lists.

enumerate A package that is superseded now.

```
\usepackage{paralist}
Do you like inline description lists? Try them out!
\begin{compactdesc}
\item[paralist] A useful package as it supports
 \begin{inparadesc} \item[compact\ldots] environments
  that have zero vertical space, \item[aspara\ldots]
  environments  formatted as paragraphs, and
  \item[inpara\ldots] environments as inline lists.
 \end{inparadesc}
\item[enumerate] A package that is superseded now.
\end{compactdesc}
```

3-3-15

Adjusting defaults

Besides providing these useful new environments the paralist package lets you customize the default settings of enumerated and itemized lists.

You can specify the default labels for different levels of itemized lists with the help of the `\setdefaultitem` declaration. It takes four arguments (as four levels of nesting are possible). In each argument you specify the desired label (just as you do with the optional argument on the environment itself) or, if you are satisfied with the default for the given level, you specify an empty argument.

- Outer level is using the default label.
    - On the second level we use again a bullet.
        - ⋆ And on the third level a star.

```
\usepackage{paralist} \setdefaultitem{}{\textbullet}{$\star$}{}
\begin{compactitem}
\item Outer level is using the default label.
  \begin{compactitem}
  \item On the second level we use again a bullet.
    \begin{compactitem}
    \item And on the third level a star.
    \end{compactitem}
  \end{compactitem}
\end{compactitem}
```

3-3-16

The changed defaults apply to all subsequent itemized environments. Normally, such a declaration is placed into the preamble, but you can also use it to change the defaults mid-document. In particular, you can define environments that contain a `\setdefaultitem` declaration which would then apply only to that particular environment—and to lists nested within its body.

You will probably not be surprised to learn that a similar declaration exists for enumerations. By using `\setdefaultenum` you can control the default look and feel of such environments. Again, there are four arguments corresponding to the four levels. In each you either specify your label definition (using the syntax explained earlier) or you leave it empty to indicate that the default for this level should be used.

1) All levels get a closing parenthesis in this example.
a) Lowercase letters here.
i) Roman numerals here.
ii) Really!

3-3-17

```
\usepackage{paralist}  \setdefaultenum{1)}{a)}{i)}{A)}
\begin{compactenum}
\item All levels get a closing parenthesis in this example.
  \begin{compactenum}
  \item Lowercase letters here.
    \begin{compactenum}
    \item Roman numerals here. \item Really!
    \end{compactenum}
  \end{compactenum}
\end{compactenum}
```

There is also the possibility of adjusting the indentation for the various list levels using the declaration `\setdefaultleftmargin`. However, this command has six arguments (there are a total of six list levels in the standard LaTeX classes), each of which takes either a dimension denoting the increase of the indention at that level or an empty argument indicating to use the current value as specified by the class or elsewhere. Another difference from the previous declarations is that in this case we are talking about the absolute list levels and not about relative levels related to either enumerations or itemizations (which can be mixed). Compare the next example with the previous one to see the difference.

1) All levels get a closing parenthesis in this example.
a) Lowercase letters here.
i) Roman numerals here.
ii) Really!

3-3-18

```
\usepackage{paralist}
\setdefaultenum{1)}{a)}{i)}{A)}
\setdefaultleftmargin{\parindent}{\parindent}
                     {\parindent}{}{}{}
\begin{compactenum}
\item All levels get a closing parenthesis in this example.
  \begin{compactenum}
  \item Lowercase letters here.
    \begin{compactenum}
    \item Roman numerals here. \item Really!
    \end{compactenum}
  \end{compactenum}
\end{compactenum}
```

By default, enumeration and itemized lists set their labels flush right. This behavior can be changed with the help of the option flushleft.

As described earlier, the label of the standard description list can be adjusted by modifying \descriptionlabel, which is also responsible for formatting the label in a compactdesc environment. With inparadesc and asparadesc, however, a different command, \paradescriptionlabel, is used for this purpose. As these environments handle their labels in slightly different ways, they do not need adjustments involving \labelsep (see page 147). Thus, its default definition is simply:

```
\newcommand*\paradescriptionlabel[1]{\normalfont\bfseries #1}
```

Finally, the paralist package supports the use of a configuration file named paralist.cfg, which by default is loaded if it exists. You can prevent this by specifying the option nocfg.

3.3.3 amsthm—Providing headed lists

The term "headed lists" describes typographic structures that, like other lists such as quotations, form a discrete part of a section or chapter and whose start and finish, at least, must be clearly distinguished. This is typically done by adjusting the vertical space at the start or adding a rule, and in this case also by including some kind of heading, similar to a sectioning head. The end may also be distinguished by a rule or other symbol, maybe within the last paragraph, and by extra vertical space.

Another property that distinguishes such lists is that they are often numbered, using either an independent system or in conjunction with the sectional numbering.

Perhaps one of the more fruitful sources of such "headed lists" is found in the so-called "theorem-like" environments. These had their origins in mathematical papers and books but are equally applicable to a wide range of expository material, as examples and exercises may take this form whether or not they contain mathematical material.

Because their historical origins lie in the mathematical world, we choose to describe the amsthm package [7] by Michael Downes from the American Mathematical Society (AMS) as a representative of this kind of extension.[1] This package provides an enhanced version of standard LaTeX's \newtheorem declaration for specifying theorem-like environments (headed lists).

As in standard LaTeX, environments declared in this way take an optional argument in which extra text, known as "notes", can be added to the head of the environment. See the example below for an illustration.

[1] When the amsthm package is used with a non-AMS document class and with the amsmath package, amsthm must be loaded *after* amsmath. The AMS document classes incorporate both packages.

```
\newtheorem*{name}{heading}
```

The `\newtheorem` declaration has two mandatory arguments. The first is the environment name that the author would like to use for this element. The second is the heading text.

If `\newtheorem*` is used instead of `\newtheorem`, no automatic numbers will be generated for the environments. This form of the command can be useful if you have only one lemma or exercise and do not want it to be numbered; it is also used to produce a special named variant of one of the common theorem types.

Lemma 1 (Main). *The LaTeX Companion complements any LaTeX introduction.*

Mittelbach's Lemma. *The LaTeX Companion contains packages from all application areas.* 3-3-19

```
\usepackage{amsthm}
\newtheorem{lem}{Lemma}
\newtheorem*{ML}{Mittelbach's Lemma}
\begin{lem}[Main] The \LaTeX{} Companion
  complements any \LaTeX{} introduction.
\end{lem}
\begin{ML} The \LaTeX{} Companion contains
  packages from all application areas.
\end{ML}
```

In addition to the two mandatory arguments, `\newtheorem` has two mutually exclusive optional arguments. They affect the sequencing and hierarchy of the numbering.

```
\newtheorem{name}[use-counter]{heading}
\newtheorem{name}{heading}[number-within]
```

By default, each kind of theorem-like environment is numbered independently. Thus, if you have lemmas, theorems, and some examples interspersed, they will be numbered something like this: Example 1, Lemma 1, Lemma 2, Theorem 1, Example 2, Lemma 3, Theorem 2. If, for example, you want the lemmas and theorems to share the same numbering sequence—Example 1, Lemma 1, Lemma 2, Theorem 3, Example 2, Lemma 4, Theorem 5—then you should indicate the desired relationship as follows:

```
\newtheorem{thm}{Theorem}     \newtheorem{lem}[thm]{Lemma}
```

The optional *use-counter* argument (value `thm`) in the second statement means that the `lem` environment should share the `thm` numbering sequence instead of having its own independent sequence.

To have a theorem environment numbered subordinately within a sectional unit—for example, to get exercises numbered Exercise 2.1, Exercise 2.2, and so on, in Section 2—put the name of the parent counter in square brackets in the final position:

```
\newtheorem{exa}{Exercise}[section]
```

With the optional argument [section], the exa counter will be reset to 0 whenever the parent counter section is incremented.

Defining the style of headed lists

The specification part of the amsthm package supports the notion of a current theorem style, which determines the formatting that will be set up by a collection of \newtheorem commands.[1]

```
\theoremstyle{style}
```

The three theorem styles provided by the package are plain, definition, and remark; they specify different typographical treatments that give the environments a visual emphasis corresponding to their relative importance. The details of this typographical treatment may vary depending on the document class, but typically the plain style produces italic body text and the other two styles produce Roman body text.

To create new theorem-like environments in these styles, divide your \newtheorem declarations into groups and preface each group with the appropriate \theoremstyle. If no \theoremstyle command is given, the style used will be plain. Some examples follow:

Definition 1. A typographical challenge is a problem that cannot be solved with the help of *The LaTeX Companion*.

Theorem 2. *There are no typographical challenges.*

Remark. The proof is left to the reader.

```
\usepackage{amsthm}
\theoremstyle{plain}       \newtheorem{thm}{Theorem}
\theoremstyle{definition} \newtheorem{defn}[thm]{Definition}
\theoremstyle{remark}      \newtheorem*{rem}{Remark}
\begin{defn}
 A typographical challenge is a problem that cannot be
 solved with the help of \emph{The \LaTeX{} Companion}.
\end{defn}
\begin{thm}There are no typographical challenges.\end{thm}
\begin{rem}The proof is left to the reader.\end{rem}
```

3-3-20

Note that the fairly obvious choice of "def" for the name of a "Definition" environment does not work, because it conflicts with the existing low-level TEX command \def.

Number swapping

A fairly common style variation for theorem heads is to have the theorem number on the left, at the beginning of the heading, instead of on the right. As this variation is usually applied across the board regardless of individual \theoremstyle changes, swapping numbers is done by placing a \swapnumbers declaration at the beginning of the list of \newtheorem statements that should be affected.

[1] This was first introduced in the now-superseded theorem package by Frank Mittelbach.

Advanced customization

More extensive customization capabilities are provided by the package through the `\newtheoremstyle` declaration and through a mechanism for using package options to load custom theorem style definitions.

```
\newtheoremstyle{name}{space-above}{space-below}{body-font}{indent}
        {head-font}{head-after-punct}{head-after-format}{head-full-spec}
```

To set up a new style of "theorem-like" headed list, use this declaration with the nine mandatory arguments described below. For many of these arguments, if they are left empty, a default is used as listed here.

name The name used to refer to the new style.

space-above The vertical space above the headed list, a rubber length (default `\topsep`).

space-below The vertical space below the headed list, a rubber length (default `\topsep`).

body-style A declaration of the font and other aspects of the style to use for the text in the body of the list (default `\normalfont`).

indent The extra indentation of the first line of the list, a non-rubber length (default is no extra indent).

head-style A declaration of the font and other aspects of the style to use for the text in the head of the list (default `\normalfont`).

head-after-punct The text (typically punctuation) to be inserted after the head text, including any note text.

head-after-space The horizontal space to be inserted after the head text and "punctuation", a rubber length. It cannot be completely empty. As two very special cases it can contain either a single space character to indicate that just a normal interword space is required or, more surprisingly, just the command `\newline` to indicate that a new line should be started for the body of the list.

head-full-spec A non-empty value for this argument enables a complete specification of the setting of the head itself to be supplied; an empty value means that the layout of the "plain" theorem style is used. See below for further details.

Any extra set-up code for the whole environment is best put into the *body-style* argument, although care needs to be taken over how it will interact with what is set up automatically. Anything that applies only to the head can be put in *head-style*.

In the example below we define a `break` theorem style, which starts a new line after the heading. The heading text is set in bold sans serif, followed by a colon and outdented into the margin by 12 pt. Since the book examples are typeset in a very small measure, we added `\RaggedRight` to the *body-style* argument.

```
\usepackage{ragged2e,amsthm}
\newtheoremstyle{break}%
  {9pt}{9pt}%                Space above and below
  {\itshape\RaggedRight}%  Body style
  {-12pt}%                   Heading indent amount
  {\sffamily\bfseries}{:}% Heading font and punctuation after it
  {\newline}%         Space after heading (\newline = linebreak)
  {}%                 Head spec (empty = same as ‘plain’ style)
\theoremstyle{break}
\newtheorem{exa}{Exercise}
\begin{exa}[Active author]
   Find the author responsible for the largest number of
   packages  described in The \LaTeX{} Companion.
\end{exa}
```

Exercise 1 (Active author):
Find the author responsible for the largest number of packages described in The LaTeX Companion.

3-3-21

Specifying the heading format

The *head-full-spec* argument, if non-empty, becomes the definition part of an internal command that is used to typeset the (up to) three bits of information contained in the head of a theorem-like environment: its number (if any), its name, and any extra notes supplied by the author when using the environment. Thus, it should contain references to three arguments that will then be replaced as follows:

- `#1` The fixed text that is to be used in the head (for example, “Exercises”), It comes from the `\newtheorem` used to declare an environment.
- `#2` A representation of the number of the element, if it should be numbered. It is conventionally left empty if the environment should not be numbered.
- `#3` The text for the optional note, from the environment’s optional argument.

Assuming all three parts are present, the contents of the *head-full-spec* argument could look as follows:

```
#1 #2 \textup{(#3)}
```

Although you are free make such a declaration, it is normally best not to use these arguments directly as this might lead to unwanted extra spaces if, for example, the environment is unnumbered.

To account for this extra complexity, the package offers three additional commands, each of which takes one argument: `\thmname`, `\thmnumber`, and `\thmnote`. These three commands are redefined at each use of the environment so as to process their arguments in the correct way. The default for each of them is simply to “typeset the argument”. Nevertheless, if, for example, the particular occurrence is unnumbered, then \thmnumber gets redefined to do no typesetting. Thus, a better definition for the *head-full-spec* argument would be

```
\thmname{#1}\thmnumber{ #2}\thmnote{ \textup{(#3)}}
```

which corresponds to the set-up used by the default plain style. Note the spaces within the last two arguments: they provide the interword spaces needed to separate the parts of the typeset head but, because they are inside the arguments, they are present only if that part of the head is typeset.

In the following example we provide a "Theorem" variation in which the whole theorem heading has to be supplied as an optional note, such as for citing theorems from other sources.

```
\usepackage{amsthm}
\newtheoremstyle{citing}%  Name
  {3pt}{3pt}%              Space above and below
  {\itshape}%              Body font
  {\parindent}{\bfseries}% Heading indent and font
  {.}%                     Punctuation after heading
  { }%   Space after head (" " = normal interword space)
  {\thmnote{#3}}%          Typeset note only, if present
\theoremstyle{citing}   \newtheorem*{varthm}{}
```

Theorem 3.16 in [87]. *By focusing on small details, it is possible to understand the deeper significance of a passage.*

3-3-22

```
\begin{varthm}[Theorem 3.16 in \cite{Knuth90}]
By focusing on small details, it is possible to
understand the deeper significance of a passage.
\end{varthm}
```

Proofs and the QED symbol

Of more specifically mathematical interest, the package defines a proof environment that automatically adds a "QED symbol" at the end. This environment produces the heading "Proof" with appropriate spacing and punctuation.[1]

An optional argument of the proof environment allows you to substitute a different name for the standard "Proof". If you want the proof heading to be, for example, "Proof of the Main Theorem", then put this in your document:

```
\begin{proof}[Proof of the Main Theorem]
  ...
\end{proof}
```

A "QED symbol" (default □) is automatically appended at the end of a proof environment. To substitute a different end-of-proof symbol, use \renewcommand to redefine the command \qedsymbol. For a long proof done as a subsection or

[1] The proof environment is primarily intended for short proofs, no more than a page or two in length. Longer proofs are usually better done as a separate \section or \subsection in your document.

section, you can obtain the symbol and the usual amount of preceding space by using the command \qed where you want the symbol to appear.

Automatic placement of the QED symbol can be problematic if the last part of a proof environment is, for example, tabular or a displayed equation or list. In that case put a \qedhere command at the somewhat earlier place where the QED symbol should appear; it will then be suppressed from appearing at the logical end of the proof environment. If \qedhere produces an error message in an equation, try using \mbox{\qedhere} instead.

Proof (sufficiency). This proof involves a list:

1. because the proof comes in two parts —
2. — we need to use \qedhere. □

```
\usepackage{amsthm}
\begin{proof}[Proof (sufficiency)]
This proof involves a list:
\begin{enumerate}
 \item because the proof comes in two parts ---
 \item --- we need to use \verb|\qedhere|. \qedhere
\end{enumerate}
\end{proof}
```

3-3-23

3.3.4 Making your own lists

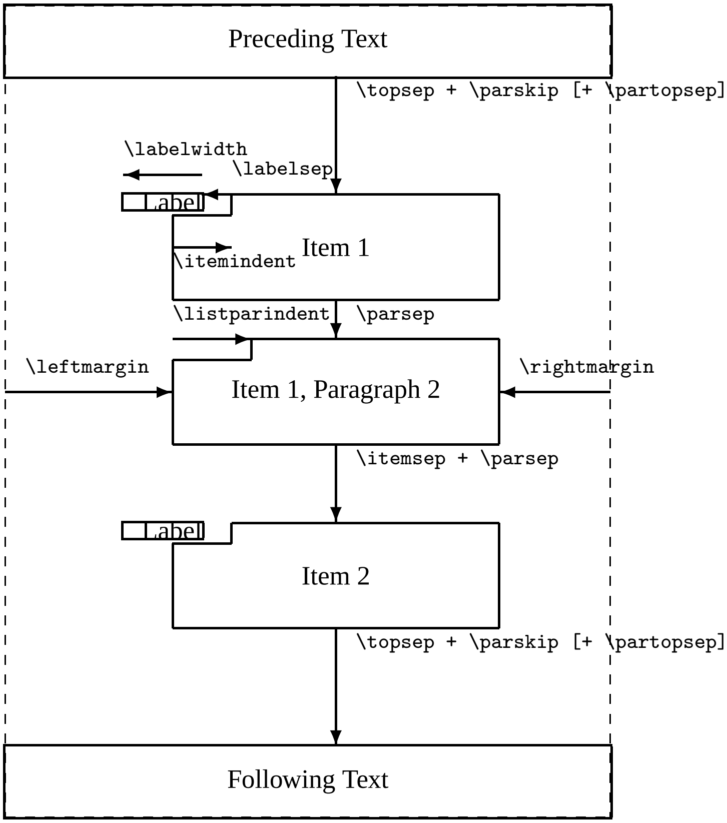
Most lists in LaTeX, including those we have seen previously, are internally built using the generic list environment. It has the following syntax:

\begin{list}{*default-label*}{*decls*} *item-list* \end{list}

The argument *default-label* is the text to be used as a label when an \item command is found without an optional argument. The second argument, *decls*, can be used to modify the different geometrical parameters of the list environment, which are shown schematically in Figure 3.3 on the next page.

The default values of these parameters typically depend on the type size and the level of the list. Those being vertically oriented are rubber lengths, meaning that they can stretch or shrink. They are set by the list environment as follows: upon entering the environment the internal command \@list⟨*level*⟩ is executed, where ⟨*level*⟩ is the list nesting level represented as a Roman numeral (e.g., \@listi for the first level, \@listii for the second, \@listiii for the third, and so on). Each of these commands, defined by the document class, holds appropriate settings for the given level. Typically, the class contains separate definitions for each major document size available via options. For example, if you select the option 11pt, one of its actions is to change the list defaults. In the standard classes this is done by loading the file size11.clo, which contains the definitions for the 11 pt document size.

In addition, most classes contain redefinitions of \@listi (i.e., first-level list defaults) within the size-changing commands \normalsize, \small, and \footnotesize, the assumption being that one might have lists within “small”



3-3-24

`\topsep` rubber space between first item and preceding paragraph.

`\partopsep` extra rubber space added to `\topsep` when environment starts a new paragraph.

`\itemsep` rubber space between successive items.

`\parsep` rubber space between paragraphs within an item.

`\leftmargin` space between left margin of enclosing environment (or of page if top-level list) and left margin of this list. Must be non-negative. Its value depends on the list level.

`\rightmargin` similar to `\leftmargin` but for the right margin. Its value is usually 0pt.

`\listparindent` extra indentation at beginning of every paragraph of a list except the one started by `\item`. Can be negative, but is usually 0pt.

`\itemindent` extra indentation added to the horizontal indentation of the text part of the first line of an item. The starting position of the label is calculated with respect to this reference point by subtracting the values of `\labelsep` and `\labelwidth`. Its value is usually 0pt.

`\labelwidth` the nominal width of the box containing the label. If the natural width of the label is ≤`\labelwidth`, then by default the label is typeset flush right inside a box of width `\labelwidth`. Otherwise, a box of the natural width is employed, which causes an indentation of the text on that line. It is possible to modify the way the label is typeset by providing a definition for the `\makelabel` command.

`\labelsep` the space between the end of the label box and the text of the first item. Its default value is 0.5 em.

Figure 3.3: Parameters used by the `list` environment

or “footnote-sized” text. However, since this is a somewhat incomplete set-up, strange effects are possible if you

- Use nested lists in such small sizes (the nested lists get the standard defaults intended for `\normalsize`),
- Jump from `\small` or `\footnotesize` directly to a large size, such as `\huge` (a first-level list now inherits the defaults from the small size, since in this set-up `\huge` does not reset the list defaults).

With a more complex set-up these defects could be mended. However, since the simpler set-up works well in most practical circumstances, most classes provide only this restricted support.

Global changes are difficult

Because of this size- and nesting-dependent set-up for the list parameters, it is not possible to change any of them globally in the preamble of your document. For global changes you have to provide redefinitions for the various `\@list..` commands discussed above or select a different document class.

Page breaking around lists

Page breaking around and within a list structure is controlled by three TeX counters: `\@beginparpenalty` (for breaking before the list), `\@itempenalty` (for breaking before an item within the list), and `\@endparpenalty` (for breaking the page after a list). By default, all three are set to a slightly negative value, meaning that it is permissible (and even preferable) to break a page in these places compared to other break points. However, this outcome may not be appropriate. You may prefer to discourage or even prevent page breaks directly before a list. To achieve this, assign a high value to `\@beginparpenalty` (10000 or more prohibits the break in all circumstances), for example:

```
\makeatletter  \@beginparpenalty=9999  \makeatother
```

TeX counters need this unusual assignment form and since all three contain an `@` sign in their name, you have to surround them with `\makeatletter` and `\makeatother` if the assignment is done in the preamble.

Many environments are implemented as lists

It is important to realize that such a setting is global to all environments based on the generic `list` environment (unless it is made in the *decls* argument) and that several LaTeX environments are defined with the help of this environment (for example, `quote`, `quotation`, `center`, `flushleft`, and `flushright`). These environments are “lists” with a single item, and the `\item[]` command is specified in the environment definition. The main reason for them to be internally defined as lists is that they then share the vertical spacing with other display objects and thus help achieve a uniform layout.

As an example, we can consider the `quote` environment, whose definition gives the same left and right margins. The simple variant `Quote`, shown below, is identical to `quote` apart from the double quote symbols added around the text. Note the special precautions, which must be taken to eliminate undesirable white space in front of (`\ignorespaces`) and following (`\unskip`) the text. We also placed the quote characters into boxes of zero width to make the quotes hang into the margin. (This trick is worth remembering: if you have a zero-width box and align the contents with the right edge, they will stick out to the left.)

… text before.

“Some quoted text, followed by more quoted text.”

Text following …

3-3-25

```
\newenvironment{Quote}%
 {\begin{list}{}%
              {\setlength\rightmargin{\leftmargin}}%
  \item[]\makebox[0pt][r]{``}\ignorespaces}%
 {\unskip\makebox[0pt][l]{''}\end{list}}
\ldots\ text before.
\begin{Quote}
   Some quoted text, followed by more quoted text.
\end{Quote}
Text following \ldots
```

In the remainder of this section we will construct a number of different “description” lists, thereby explaining the various possibilities offered by the generic `list` environment. We start by looking at the default definition of the `description` environment as it can be found in LaTeX’s standard classes such as article or report.[1]

```
    \newenvironment{description}
        {\begin{list}{}{\setlength\labelwidth{0pt}%
                        \setlength\itemindent{-\leftmargin}%
                        \let\makelabel\descriptionlabel}}
        {\end{list}}
```

To understand the reasoning behind this definition recall Figure 3.3 on page 145, which explains the relationship between the various list parameters. The parameter settings start by setting `\labelwidth` to zero, which means that we do not reserve any space for the label. Thus, if the label is being typeset, it will move the text of the first line to the right to get the space it needs. Then the `\itemindent` parameter is set to the negation of `\leftmargin`. As a result, the starting point for the first text line is moved to the enclosing margin but all turnover lines are still indented by `\leftmargin`. The last declaration makes `\makelabel` identical to the command `\descriptionlabel`. The command `\makelabel` is called by the `list` environment whenever it has to format an item label. It takes one argument (the label) and is supposed to produce a typeset version of that argument. So the final task to finish the definition of the `description` environment is to provide a suitable definition for `\descriptionlabel`. This indirection is useful because it allows us to change the label formatting without modifying the rest of the environment definition.

How should `\descriptionlabel` be defined? It has to provide the formatting for the label. With the standard `description` environment this label is supposed

[1] If you look into `article.cls` or `report.cls` you will find a slightly optimized coding that uses, for example, low-level assignments instead of `\setlength`. However, conceptually, the definitions are identical.

to be typeset in boldface. But recall that the label is separated from the following text by a space of width `\labelsep`. Due to the parameter settings given above this text starts at the outer margin. Thus, without correction our label would end up starting in the margin (by the width of `\labelsep`). To prevent this outcome the standard definition for the `\descriptionlabel` command has the following curious definition, in that it first moves to the right and then typesets the label:

```
\newcommand*\descriptionlabel[1]
    {\hspace{\labelsep}\normalfont\bfseries #1}
```

To remove this dependency, one would need to change the setting of `\itemindent` to already take the `\labelsep` into account, which in itself would not be difficult. You may call this behavior an historical artifact, but many documents rely on this somewhat obscure feature. Thus, it is difficult to change the setting in the LaTeX kernel without breaking those documents.

With the parameter settings of the standard `description` environment, in case of short labels the text of the first line starts earlier than the text of remaining lines. If we always want a minimal indentation we can try a definition similar to the one in the following example, where we set `\labelwidth` to `40pt` and `\leftmargin` to `\labelwidth` plus `\labelsep`. This means that `\makelabel` has to concern itself only with formatting the label. However, given that we now have a positive nominal label width, we need to define what should happen if the label is small. By using `\hfill` we specify where extra white space should be inserted.

Description: Returns from a function. If issued at top level, the interpreter simply terminates, just as if end of input had been reached.

Errors: None.

Return values:
Any arguments in effect are passed back to the caller.

```
\usepackage{calc}
\newenvironment{Description}
   {\begin{list}{}{\let\makelabel\Descriptionlabel
      \setlength\labelwidth{40pt}%
      \setlength\leftmargin{\labelwidth+\labelsep}}}%
   {\end{list}}
\newcommand*\Descriptionlabel[1]{\textsf{#1:}\hfil}
\begin{Description}
\item[Description]
  Returns from a function. If issued at top level,
  the interpreter simply terminates, just as if
  end of input had been reached.
\item[Errors]  None.
\item[Return values]
  \mbox{}\\
  Any arguments in effect are passed back to the
  caller.
\end{Description}
```

3-3-26

This example shows a typical problem with description-like lists when the text in the label (*term*) is wider than the width of the label. Our definition lets the text of the term continue into the text of the *description* part. This is often not desired, and to improve the visual appearance of the list we have started one of the description parts on the next line. A new line was forced by putting an empty box on the same line, followed by the ‘\\’ command.

In the remaining part of this section various possibilities for controlling the width and mutual positioning of the term and description parts will be investigated. The first method changes the width of the label. The environment is declared with an argument specifying the desired width of the label field (normally chosen to be the widest term entry). Note the redefinition of the `\makelabel` command where you specify how the label will be typeset. As this redefinition is placed inside the definition[1] of the `altDescription` environment, the argument placeholder character # must be escaped to ## to signal LaTeX that you are referring to the argument of the `\makelabel` command, and not to the argument of the outer environment. In such a case, `\labelwidth` is set to the width of the environment’s argument after it is processed by `\makelabel`. This way formatting directives for the label that might change its width are taken into account.

Description: Returns from a function. If issued at top level, the interpreter simply terminates, just as if end of input had been reached.

Errors: None.

Return values: Any arguments in effect are passed back to the caller.

3-3-27

```
\usepackage{calc}
\newenvironment{altDescription}[1]
  {\begin{list}{}%
    {\renewcommand\makelabel[1]{\textsf{##1:}\hfil}%
     \settowidth\labelwidth{\makelabel{#1}}%
     \setlength\leftmargin{\labelwidth+\labelsep}}}%
  {\end{list}}
\begin{altDescription}{Return values}
\item[Description]
   Returns from a function.  If issued at top level,
   the interpreter simply terminates, just as if end
   of input had been reached.
\item[Errors]
   None.
\item[Return values]
   Any arguments in effect are passed back to the
   caller.
\end{altDescription}
```

A similar environment (but using an optional argument) is shown in Example A-1-9 on page 850. However, having several lists with varying widths for the label field on the same page might look typographically unacceptable. Evaluating the width of the term is another possibility that avoids this problem. If the width is wider than `\labelwidth`, an additional empty box is appended with the effect that the description part starts on a new line. This matches the conventional method for displaying options in UN*X manuals.

To illustrate this method we reuse the `Description` environment defined

[1] This is done for illustration purposes. Usually the solution involving an external name is preferable, as with `\Descriptionlabel` in Example 3-3-26 on the preceding page.

in Example 3-3-26 but provide a different definition for the `\Descriptionlabel` command as follows:

Description:
: Returns from a function. If issued at top level, the interpreter simply terminates, just as if end of input had been reached.

Errors: None.

Return values:
: Any arguments in effect are passed back to the caller.

```
\usepackage{calc,ifthen}   \newlength{\Mylen}
% definition of Description environment as before
\newcommand*\Descriptionlabel[1]{%
  \settowidth\Mylen{\textsf{#1:}}%   determine width
  \ifthenelse{\lengthtest{\Mylen > \labelwidth}}%
      {\parbox[b]{\labelwidth}%      term > labelwidth
         {\makebox[0pt][l]{\textsf{#1:}}\\}}%
      {\textsf{#1:}}%                term < labelwidth
  \hfill}
\begin{Description}
\item[Description]  Returns from a function.
 If issued at top level, the interpreter simply
 terminates, just as if end of input had been reached.
\item[Errors]   None.
\item[Return values]
 Any arguments in effect are passed back to the caller.
\end{Description}
```

3-3-28

The definition of `\Descriptionlabel` sets the length variable `\Mylen` equal to the width of the label. It then compares that length with `\labelwidth`. If the label is smaller than `\labelwidth`, then it is typeset on the same line as the description term. Otherwise, it is typeset in a zero-width box with the material sticking out to the right as far as needed. It is placed into a bottom-aligned `\parbox` followed by a forced line break so that the description term starts one line lower. This somewhat complicated maneuver is necessary because `\makelabel`, and thus `\Descriptionlabel`, are executed in a strictly horizontal context in which vertical spaces or `\\` commands have no effect.

Yet another possibility is to allow multiple-line labels.

Descrip-
tion: Returns from a function. If issued at top level, the interpreter simply terminates, just as if end of input had been reached.

Errors: None.

Return
values: Any arguments in effect are passed back to the caller.

```
\usepackage{calc}
% definition of Description environment as before
\newcommand*\Descriptionlabel[1]
   {\raisebox{0pt}[1ex][0pt]%
      {\makebox[\labelwidth][l]%
         {\parbox[t]{\labelwidth}%
                    {\hspace{0pt}\textsf{#1:}}}}}
\begin{Description}
\item[Description]  Returns from a function.
 If issued at top level, the interpreter simply
 terminates, just as if end of input had been reached.
\item[Errors]   None.
\item[Return\\values]
 Any arguments in effect are passed back to the caller.
\end{Description}
```

3-3-29

In the previous example, we once again used the Description environment as a basis, with yet another redefinition of the \Descriptionlabel command. The idea here is that large labels may be split over several lines. Certain precautions have to be taken to allow hyphenation of the first word in a paragraph, and therefore the \hspace{0pt} command is introduced in the definition. The material gets typeset inside a paragraph box of the correct width \labelwidth, which is then top and left aligned into a box that is itself placed inside a box with a height of 1 ex and no depth. In this way, LaTeX does not realize that the material extends below the first line.

The final example deals with the definition of enumeration lists. An environment with an automatically incremented counter can be created by including a \usecounter command in the declaration of the list environment. This function is demonstrated with the Notes environment, which produces a sequence of notes. In this case, the first parameter of the list environment is used to provide the automatically generated text for the term part.

After declaring the notes counter, the default label of the Notes environment is defined to consist of the word NOTES in small caps, followed by the value of the notes counter, using as its representation an Arabic numeral followed by a dot. Next \labelsep is set to a relatively large value and \itemindent, \leftmargin, and \labelwidth are adjusted in a way such that the label nevertheless starts out at the left margin. Finally, the already-mentioned \usecounter declaration ensures that the notes counter is incremented for each \item command.

```
\newcounter{notes}
\newenvironment{Notes}
    {\begin{list}{\textsc{Note} \arabic{notes}.}%
                 {\setlength\labelsep{10pt}%
                  \setlength\itemindent{10pt}%
                  \setlength\leftmargin{0pt}%
                  \setlength\labelwidth{0pt}%
                  \usecounter{notes}}}%
    {\end{list}}
\begin{Notes}
\item This is the text of the first note item.
      Some more text for the first note item.
\item This is the text of the second note item.
      Some more text for the second note item.
\end{Notes}
```

NOTE 1. This is the text of the first note item. Some more text for the first note item.

NOTE 2. This is the text of the second note item. Some more text for the second note item.

3-3-30

3.4 Simulating typed text

It is often necessary to display information verbatim—that is, “as entered at the terminal”. This ability is provided by the standard LaTeX environment verbatim. However, to guide the reader it might be useful to highlight certain textual strings in a particular way, such as by numbering the lines. Over time a number of packages have appeared that addressed one or the other extra feature—unfortunately, each with its own syntax.

In this section we will review a few such packages. Since they have been used extensively in the past, you may come across them in document sources on the Internet or perhaps have used them yourself in the past. But we then concentrate on the package fancyvrb written by Timothy Van Zandt, which combines all such features and many more under the roof of a single, highly customizable package.

This coverage is followed by a discussion of the listings package, which provides a versatile environment in which to pretty print computer listings for a large number of computer languages.

3.4.1 Simple verbatim extensions

The package alltt (by Leslie Lamport) defines the `alltt` environment. It acts like a `verbatim` environment except that the backslash "\" and braces "{" and "}" retain their usual meanings. Thus, other commands and environments can appear inside an `alltt` environment. A similar functionality is provided by the fancyvrb environment keyword `commandchars` (see page 161).

`One can have font changes, like`
`emphasized text.`
`Some special characters: # $ % ^ & ~ _`

```
\usepackage{alltt}
\begin{alltt}
One can have font changes, like
\emph{emphasized text}.
Some special characters: # $ % ^ & ~ _
\end{alltt}
```

3-4-1

In documents where a lot of `\verb` commands are needed the source soon becomes difficult to read. For this reason the doc package, described in Chapter 14, introduces a shortcut mechanism that lets you use a special character to denote the start and stop of verbatim text, without having to repeatedly write `\verb` in front of it. This feature is also available in a stand-alone package called shortvrb. With fancyvrb the same functionality is provided, unfortunately using a slightly different syntax (see page 167).

The use of `\MakeShortVerb` can make sources much more readable. And with the declaration `\DeleteShortVerb{\|}` we can return the | character back to normal.

```
\usepackage{shortvrb}
\MakeShortVerb{\|}
The use of |\MakeShortVerb| can make sources
much more readable.
\DeleteShortVerb{\|}\MakeShortVerb{\+}
And with the declaration +\DeleteShortVerb{\|}+
we can return the +|+ character back to normal.
```

3-4-2

The variant form, `\MakeShortVerb*`, implements the same shorthand mechanism for the `\verb*` command. This is shown in the next example.

3-4-3

Instead of ␣ we can now write ␣.

```
\usepackage{shortvrb}  \MakeShortVerb*{\+}
Instead of \verb*/ / we can now write + +.
```

The package verbatim (by Rainer Schöpf) reimplements the LaTeX environments `verbatim` and `verbatim*`. One of its major advantages is that it allows arbitrarily long verbatim texts, something not possible with the basic LaTeX versions of the environments. It also defines a `comment` environment that skips all text between the commands `\begin{comment}` and `\end{comment}`. In addition, the package provides hooks to implement user extensions for defining customized verbatim-like environments.

A few such extensions are realized in the package moreverb (by Angus Duggan). It offers some interesting verbatim-like commands for writing to and reading from files as well as several environments for the production of listings and dealing with tab characters. All of these extensions are also available in a consistent manner with the fancyvrb package, so here we only give a single example to show the flavor of the syntax used by the moreverb package.

Text before listing environment.

```
    The␣listing␣environment␣numbers␣the
4   lines␣in␣it.␣␣It␣takes␣an␣optional
    argument,␣which␣is␣the␣step␣between
6   numbered␣lines␣(line␣1␣is␣always
    numbered␣if␣present),␣and␣a␣required
8   argument,␣which␣is␣the␣starting␣line.
    The␣star␣form␣makes␣blanks␣visible.
```

Text between listing environments.

```
10  This listingcont environment continues
    where the previous listing environment
12  left off.  Both the listing and
    listingcont environments expand tabs
14  with a default tab width of 8.
```

3-4-4

Text following listing environments.

```
\usepackage{verbatim,moreverb}
Text before listing environment.
\begin{listing*}[2]{3}
The listing environment numbers the
lines in it.  It takes an optional
argument, which is the step between
numbered lines (line 1 is always
numbered if present), and a required
argument, which is the starting line.
The star form makes blanks visible.
\end{listing*}
Text between listing environments.
\begin{listingcont}
This listingcont environment continues
where the previous listing environment
left off.  Both the listing and
listingcont environments expand tabs
with a default tab width of 8.
\end{listingcont}
Text following listing environments.
```

3.4.2 upquote—Computer program style quoting

The Computer Modern Typewriter font that is used by default for typesetting “verbatim” is a very readable monospaced typeface. Due to its small running length it is very well suited for typesetting computer programs and similar material. See Section 7.7.4 for a comparison of this font with other monospaced typefaces.

There is, however, one potential problem when using this font to render computer program listings and similar material: most people expect to see a (right) quote in a computer listing represented with a straight quote character (i.e., ') and a left or back quote as a kind of grave accent on its own (i.e., `). The Computer Modern Typewriter font, however, displays real left and right curly quote characters (as one would expect in a normal text font). In fact, most other typewriter fonts when set up for use with LaTeX follow this pattern. This produces somewhat unconventional results that many people find difficult to understand. Consider the following example, which shows the standard behavior for three major typewriter fonts: LuxiMono, Courier, and Computer Modern Typewriter.

```
TEST=‘ls -l |awk ’{print $3}’‘
TEST=‘ls -l |awk ’{print $3}’‘
TEST=‘ls -l |awk ’{print $3}’‘
```

```
\usepackage[scaled=0.85]{luximono}
\raggedright
\verb+TEST=‘ls -l |awk ’{print $3}’‘+
\par \renewcommand\ttdefault{pcr}
\verb+TEST=‘ls -l |awk ’{print $3}’‘+
\par \renewcommand\ttdefault{cmtt}
\verb+TEST=‘ls -l |awk ’{print $3}’‘+
```

3-4-5

This behavior can be changed by loading the package upquote (written by Michael Covington), which uses the glyphs `\textasciigrave` and `\textquotesingle` from the textcomp package instead of the usual left and right curly quote characters within `\verb` or the `verbatim` environment. Normal typewriter text still uses the curly quotes, as shown in the last line of the example.

```
TEST=`ls -l |awk '{print $3}'`
TEST=`ls -l |awk '{print $3}'`
TEST=`ls -l |awk '{print $3}'`
but ‘text’ is unaffected!
```

```
\usepackage[scaled=0.85]{luximono}
\usepackage{upquote}
\raggedright
\verb+TEST=‘ls -l |awk ’{print $3}’‘+
\par \renewcommand\ttdefault{pcr}
\verb+TEST=‘ls -l |awk ’{print $3}’‘+
\par \renewcommand\ttdefault{cmtt}
\verb+TEST=‘ls -l |awk ’{print $3}’‘+
\par \texttt{but ‘text’ is unaffected!}
```

3-4-6

The package works well together with “verbatim” extensions as described in this chapter, except for the listings package; it conflicts with the scanning mechanism of that package. If you want this type of quoting with listings simply use the `\lstset` keyword `upquote`.

```
TEST=`ls␣−l␣|awk␣'{print␣$3}'`
```

```
\usepackage{textcomp}
\usepackage{listings} \lstset{upquote}
\begin{lstlisting}[language=ksh]
TEST=‘ls -l |awk ’{print $3}’‘
\end{lstlisting}
```

3-4-7

3.4.3 fancyvrb—Highly customizable verbatim environments

The fancyvrb package by Timothy Van Zandt (these days maintained by Denis Girou and Sebastian Rahtz) offers a highly customizable set of environments and commands to typeset and manipulate verbatim text.

It works by parsing one line at a time from an environment or a file (a concept pioneered by the verbatim package), thereby allowing you to preprocess lines in various ways. By incorporating features found in various other packages it provides a truly universal production environment under a common set of syntax rules.

The main environment provided by the package is the `Verbatim` environment, which, if used without customization, behaves similarly to standard LaTeX's `verbatim` environment. The main difference is that it accepts an optional argument in which you can specify customization information using a key/value syntax. However, there is one restriction to bear in mind: the left bracket of the optional argument must appear on the same line as `\begin`. Otherwise, the optional argument will not be recognized but instead typeset as verbatim text.

More than 30 keywords are available, and we will discuss their use and possible values in some detail.

A number of variant environments and commands will be discussed near the end of this section as well. They also accept customization via the key/value method. Finally, we cover possibilities for defining your own variants in a straightforward way.

Customization keywords for typesetting

To manipulate the fonts used by the verbatim environments of the fancyvrb package, four environment keywords, corresponding to the four axes of NFSS, are available. The keyword `fontfamily` specifies the font family to use. Its default is `Computer Modern Typewriter`, so that when used without keywords the environments behave in similar fashion to standard LaTeX's `verbatim`. However, the value of this keyword can be any font family name in NFSS notation, such as `pcr` for `Courier` or `cmss` for Computer Modern Sans, even though the latter is not a monospaced font as would normally be used in a verbatim context. The keyword also recognizes the special values `tt`, `courier`, and `helvetica` and translates them internally into NFSS nomenclature.

Because typesetting of verbatim text can include special characters like "\" you must be careful to ensure that such characters are present in the font. This should be no problem when a font encoding such as `T1` is active, which could be loaded using the fontenc package. It is, however, not the case for LaTeX's default font encoding `OT1`, in which only some monospaced fonts, such as the default typewriter font, contain all such special characters. The type of incorrect output you might encounter is shown in the second line of the next example.

```
Family ‘tt’ is fine in OT1: \sum_{i=1}^n

But ‘helvetica’ fails in OT1: “sum˙–i=1˝ˆn

...  while it works in T1: \sum_{i=1}^n
```

```
\usepackage{fancyvrb}
\usepackage[OT1,T1]{fontenc}
\fontencoding{OT1}\selectfont
\begin{Verbatim}[fontfamily=tt]
Family ‘tt’ is fine in OT1: \sum_{i=1}^n
\end{Verbatim}
\begin{Verbatim}[fontfamily=helvetica]
But ‘helvetica’ fails in OT1: \sum_{i=1}^n
\end{Verbatim}
\fontencoding{T1}\selectfont
\begin{Verbatim}[fontfamily=helvetica]
...  while it works in T1: \sum_{i=1}^n
\end{Verbatim}
```

3-4-8

Since all examples in this book are typeset using the T1 encoding this kind of problem will not show up elsewhere in the book. Nevertheless, you should be aware of this danger. It represents another good reason to use T1 in preference to TEX's original font encoding; for a more in-depth discussion see Section 7.2.4 on page 336.

The other three environment keywords related to the font set-up are `fontseries`, `fontshape`, and `fontsize`. They inherit the current NFSS settings from the surrounding text if not specified. While the first two expect values that can be fed into `\fontseries` and `\fontshape`, respectively (e.g., `bx` for a bold extended series or `it` for an *italic* shape), the `fontsize` is special. It expects one of the higher-level NFSS commands for specifying the font size—for example, `\small`. If the relsize package is available then you could alternatively specify a change of font size relative to the current text font by using something like `\relsize{-2}`.

```
  \sum_{i=1}^n

A line of text to show the body size.

  \sum_{i=1}^n
```

```
\usepackage{relsize,fancyvrb}
\begin{Verbatim}[fontsize=\relsize{-2}]
  \sum_{i=1}^n
\end{Verbatim}
A line of text to show the body size.
\begin{Verbatim}[fontshape=sl,fontsize=\Large]
  \sum_{i=1}^n
\end{Verbatim}
```

3-4-9

A more general form for customizing the formatting is available through the environment keyword `formatcom`, which accepts any LATEX code and executes it at the start of the environment. For example, to color the verbatim text you could pass it something like `\color{blue}`. There is also the possibility to operate on each line of text by providing a suitable redefinition for the command `\FancyVerbFormatLine`. This command is executed for every line, receiving the text from the line as its argument. In the next example every second line is colored in blue, a result achieved by testing the current value of the counter FancyVerbLine. This counter is provided automatically by the environment and holds the current line number.

```
This line should become blue while
 this one will be black. And here
u can observe that gobble removes
t only blanks but any character.
```

3-4-10

```
\usepackage{ifthen,color,fancyvrb}
\renewcommand\FancyVerbFormatLine[1]
  {\ifthenelse{\isodd{\value{FancyVerbLine}}}%
             {\textcolor{blue}{#1}}{#1}}
\begin{Verbatim}[gobble=2]
  This line should become blue while
   this one will be black. And here
you can observe that gobble removes
not only blanks but any character.
\end{Verbatim}
```

As shown in the previous example the keyword gobble can be used to remove a number of characters or spaces (up to nine) from the beginning of each line. This is mainly useful if all lines in your environments are indented and you wish to get rid of the extra space produced by the indentation. Sometimes the opposite goal is desired: every line should be indented by a certain space. For example, in this book all verbatim environments are indented by 24pt. This indentation is controlled by the keyword xleftmargin. There also exists a keyword xrightmargin to specify the right indentation, but its usefulness is rather limited, since verbatim text is not broken across lines. Thus, its only visible effect (unless you use frames, as discussed below) are potentially more overfull box messages[1] that indicate that your text overfloods into the right margin. Perhaps more useful is the Boolean keyword resetmargins, which controls whether preset indentations by surrounding environments are ignored.

- Normal indentation left:

  ```
  A verbatim line of text!
  ```

- No indentation at either side:

```
A verbatim line of text!
```

3-4-11

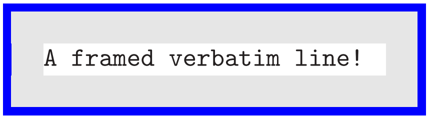
```
\usepackage{fancyvrb}
\begin{itemize}  \item  Normal indentation left:
  \begin{Verbatim}[frame=single,xrightmargin=2pc]
A verbatim line of text!
  \end{Verbatim}
  \item No indentation at either side:
  \begin{Verbatim}[resetmargins=true,
                   frame=single]
A verbatim line of text!
  \end{Verbatim}
\end{itemize}
```

The previous example demonstrates one use of the frame keyword: to draw a frame around verbatim text. By providing other values for this keyword, different-

[1] Whether overfull boxes inside a verbatim environment are shown is controlled the hfuzz keyword, which has a default value of 2pt. A warning is issued only if boxes protrude by more than the keywords's value into the margin.

looking frames can be produced. The default is none, that is, no frame. With topline, bottomline, or leftline you get a single line at the side indicated;[1] lines produces a line at top and bottom; and single, as we saw in Example 3-4-11, draws the full frame. In each case, the thickness of the rules can be customized by specifying a value via the framerule keyword (default is 0.4pt). The separation between the lines and the text can be controlled with framesep (default is the current value of \fboxsep).

If the color package is available, you can color the rules using the environment keyword rulecolor (default is black). If you use a full frame, you can also color the separation between the frame and the text via fillcolor.

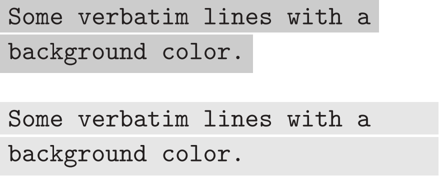


```
A framed verbatim line!
```

```
\usepackage{color,fancyvrb}
\begin{Verbatim}[frame=single,rulecolor=\color{blue},
  framerule=3pt,framesep=1pc,fillcolor=\color{yellow}]
A framed verbatim line!
\end{Verbatim}
```

3-4-12

Unfortunately, there is no direct way to fill the entire background. The closest you can get is by using \colorbox inside \FancyVerbFormatLine. But this approach will leave tiny white rules behind the lines and—without forcing the lines to be of equal length, such as via \makebox—will also result in colored blocks of different widths.



```
Some verbatim lines with a
background color.

Some verbatim lines with a
background color.
```

```
\usepackage{color,fancyvrb}
\renewcommand\FancyVerbFormatLine[1]
  {\colorbox{green}{#1}}
\begin{Verbatim}
Some verbatim lines with a
background color.
\end{Verbatim}
\renewcommand\FancyVerbFormatLine[1]
  {\colorbox{yellow}{\makebox[\linewidth][l]{#1}}}
\begin{Verbatim}
Some verbatim lines with a
background color.
\end{Verbatim}
```

3-4-13

It is possible to typeset text as part of a frame by supplying it as the value of the label keyword. If this text contains special characters, such as brackets, equals sign, or comma, you have to hide them by surrounding them with a brace group. Otherwise, they will be mistaken for part of the syntax. The text appears by default at the top, but is printed only if the frame set-up would produce a line in that position. Alternate positions can be specified by using labelposition, which accepts none, topline, bottomline, or all as values. In the last case the text is printed above and below. If the label text is unusually large you may need

[1] There is no value to indicate a line at the right side.

to increase the separation between the frame and the verbatim text by using the keyword `framesep`. If you want to cancel a previously set label string, use the value `none`—if you really need “none” as a label string, enclose it in braces.

```
 ________________________________
|                                |
|   Some verbatim text framed    |
|_______ Example code ___________|
```

3-4-14

```
\usepackage{fancyvrb}
\begin{Verbatim}[frame=single,label=\fbox{Example code},
                 framesep=5mm,labelposition=bottomline]
Some verbatim text framed
\end{Verbatim}
```

You can, in fact, provide different texts to be placed at top and bottom by surrounding the text for the top position with brackets, as shown in the next example. For this scheme to work `frame` needs to be set to either `single` or `lines`.

```
________ Start of code ________

  A line of code

_________ End of code ________
```

3-4-15

```
\usepackage{fancyvrb}
\begin{Verbatim}[frame=lines,framesep=5mm,
                 label={[Start of code]End of code}]
  A line of code
\end{Verbatim}
```

By default, the typeset output of the verbatim environments can be broken across pages by LaTeX if it does not fully fit on a single page. This is even true in cases where a frame surrounds the text. If you want to ensure that this cannot happen, set the Boolean keyword `samepage` to `true`.

The vertical spacing between lines in a verbatim environment is the same as in normal text, but if desired you can enlarge it by a factor using the keyword `baselinestretch`. Shrinking so that lines overlap is not possible. If you want to revert to the default line separation, use the string `auto` as a value.

```
This text is more or less double-spaced.

See also the discussion about the

setspace package elsewhere.
```

3-4-16

```
\usepackage{fancyvrb}
\begin{Verbatim}[baselinestretch=1.6]
This text is more or less double-spaced.
See also the discussion about the
setspace package elsewhere.
\end{Verbatim}
```

When presenting computer listings, it is often helpful to number some or all of the lines. This can be achieved by using the keyword `numbers`, which accepts `none`, `left`, or `right` as a value to control the position of the numbers. The distance between the number and the verbatim text is `12pt` by default but it can be adjusted by specifying a different value via the keyword `numbersep`. Usually, numbering restarts at 1 with each environment, but by providing an explicit number with the keyword `firstnumber` you can start with any integer value, even a negative one. Alternatively, this keyword accepts the word `last` to indicate that numbering should resume where it had stopped in the previous `Verbatim` instance.

```
1  Verbatim lines can be numbered
2  at either left or right.
```

Some intermediate text…

```
3  Continuation is possible too
4  as we can see here.
```

```
\usepackage{fancyvrb}
\begin{Verbatim}[numbers=left,numbersep=6pt]
Verbatim lines can be numbered
at either left or right.
\end{Verbatim}
Some intermediate text\ldots
\begin{Verbatim}[numbers=left,firstnumber=last]
Continuation is possible too
as we can see here.
\end{Verbatim}
```

3-4-17

Some people prefer to number only some lines, and the package caters to this possibility by providing the keyword `stepnumber`. If this keyword is assigned a positive integer number, then only line numbers being an integer multiple of that number will get printed. We already learned that the counter that is used internally to count the lines is called `FancyVerbLine`, so it comes as no surprise that the appearance of the numbers is controlled by the command `\theFancyVerbLine`. By modifying this command, special effects can be obtained; a possibility where the current chapter number is prepended is shown in the next example. It also shows the use of the Boolean keyword `numberblanklines`, which controls whether blank lines are numbered (default is `false`, i.e., to not number them).

```
     Normally empty lines in
3.2  in a verbatim will not receive
     numbers---here they do!
3.4
     Admittedly using stepnumber
3.6  with such a redefinition of
     FancyVerbLine looks a bit odd.
```

```
\usepackage{fancyvrb}
\renewcommand\theFancyVerbLine{\footnotesize
  \thechapter.\arabic{FancyVerbLine}}
\begin{Verbatim}[numbers=left,stepnumber=2,
                 numberblanklines=true]
Normally empty lines in
in a verbatim will not receive
numbers---here they do!

Admittedly using stepnumber
with such a redefinition of
FancyVerbLine looks a bit odd.
\end{Verbatim}
```

3-4-18

In some situations it helps to clearly identify white space characters by displaying all blanks as ␣. This can be achieved with the Boolean keyword `showspaces` or, alternatively, the `Verbatim*` variant of the environment.

Another white space character, the tab, plays an important rôle in some programming languages, so there may be a need to identify it in your source. This is achieved with the Boolean keyword `showtabs`. The tab character displayed is defined by the command `\FancyVerbTab` and can be redefined, as seen below. By default, tab characters simply equal eight spaces, a value that can be changed with the keyword `tabsize`. However, if you set the Boolean keyword `obeytabs` to `true`, then each tab character produces as many spaces as necessary to move to the next integer multiple of tabsize. The example input contains tabs in each line that are displayed on the right as spaces with the default tabsize of 8. Note in particular the difference between the last input and output line.

```
123456789012345678901234567890
Two       ⇥default       ⇥tabs

Two    ⇥real    ⇥tabs

Two    ▷new     ▷tabs

Using▷a ▷special tab▷size
```

3-4-19

```
\usepackage{fancyvrb}
\begin{Verbatim}[showtabs=true]
123456789012345678901234567890
Two     default tabs
\end{Verbatim}
\begin{Verbatim}[obeytabs=true,showtabs=true]
Two     real    tabs
\end{Verbatim}
\renewcommand\FancyVerbTab{$\triangleright$}
\begin{Verbatim}[obeytabs=true,showtabs=true]
Two     new     tabs
\end{Verbatim}
\begin{Verbatim}[obeytabs=true,tabsize=3,showtabs=true]
Using   a       special tab     size
\end{Verbatim}
```

If you wish to execute commands within the verbatim text, then you need one character to act as an escape character (i.e., to denote the beginning of a command name) and two characters to serve as argument delimiters (i.e., to play the rôle that braces normally play within LaTeX). Such special characters can be specified with the commandchars keyword as shown below; of course, these characters then cannot appear as part of the verbatim text. The characters are specified by putting a backslash in front of each one so as to mask any special meaning they might normally have in LaTeX. The keyword commentchar allows you to define a comment character, which will result in ignoring everything following it until and including the next new line. Thus, if this character is used in the middle of a line, this line and the next will be joined together. If you wish to cancel a previous setting for commandchars or commentchar, use the string value “none”.

```
We can emphasize text
Line with label is shown here.
```

On line 2 we see...

3-4-20

```
\usepackage{fancyvrb}
\begin{Verbatim}[commandchars=\|\[\],commentchar=\!]
We can |emph[emphasize] text
! see above (this line is invisible)
Line with label|label[linea] ! removes new line
is shown here.
\end{Verbatim}
On line~\ref{linea} we see\ldots
```

If you use \label within the verbatim environment, as was done in the previous example, it will refer to the internal line number whether or not that number is displayed. This requires the use of the commandchars keyword, a price you might consider too high because it deprives you of the use of the chosen characters in your verbatim text.

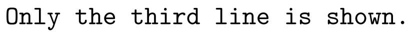
Two other keywords let you change the parsing and manipulation of verbatim data: codes and defineactive. They allow you to play some devious tricks but their use is not so easy to explain: one needs a good understanding of TEX's inner workings. If you are interested, please check the documentation provided with the fancyvrb package.

Limiting the displayed data

Normally, all lines within the verbatim environment are typeset. But if you want to display only a subset of lines, you have a number of choices. With the keywords firstline and lastline, you can specify the start line and (if necessary) the final line to typeset. Alternatively, you can specify a start and stop string to search for within the environment body, with the result that all lines between (but this time *not* including the special lines) will be typeset. The strings are specified in the macros \FancyVerbStartString and \FancyVerbStopString. To make this work you have to be a bit careful: the macros need to be defined with \newcommand* and redefined with \renewcommand*. Using \newcommand will *not* work! To cancel such a declaration is even more complicated: you have to \let the command to \relax, for example,

```
\let\FancyVerbStartString\relax
```

or ensure that your definition is confined to a group—everything else fails.


Only the third line is shown.

```
\usepackage{fancyvrb}
\newcommand*\FancyVerbStartString{START}
\newcommand*\FancyVerbStopString{STOP}
\begin{Verbatim}
  A verbatim line not shown.
START
  Only the third line is shown.
STOP
  But the remainder is left out.
\end{Verbatim}
```

3-4-21

You may wonder why one would want to have such functionality available, given that one could simply leave out the lines that are not being typeset. With an environment like Verbatim they are indeed of only limited use. However, when used together with other functions of the package that write data to files and read it back again, they offer powerful solutions to otherwise unsolvable problems.

How the book examples have been produced

For instance, all examples in this book use this method. The example body is written to a file together with a document preamble and other material, so that the resulting file will become a processable LATEX document. This document is then externally processed and included as an EPS graphic image into the book. Beside it, the sample code is displayed by reading this external file back in but displaying only those lines that lie between the strings \begin{document} and \end{document}. This accounts for the example lines you see being typeset in black. The preamble part, which is shown in blue, is produced in a similar fashion: for this the start and stop strings are redefined to include only those lines lying between the strings \StartShownPreambleCommands and \StopShownPreambleCommands. When processing the example externally, these two commands are simply no-ops; that is, they are defined by the "example" class (which is otherwise close to the article document class) to do nothing. As a consequence, the example code will always (for better or worse) correspond to the displayed result.[1]

To write data verbatim to a file the environment VerbatimOut is available. It takes one mandatory argument: the file name into which to write the data. There is, however, a logical problem if you try to use such an environment inside your own environments: the moment you start the VerbatimOut environment, everything is swallowed without processing and so the end of your environment is not recognized. As a solution the fancyvrb package offers the command \VerbatimEnvironment, which, if executed within the \begin code of your environment, ensures that the end tag of your environment will be recognized in verbatim mode and the corresponding code executed.

To read data verbatim from a file, the command \VerbatimInput can be used. It takes an optional argument similar to the one of the Verbatim environment (i.e., it accepts all the keywords discussed previously) and a mandatory argument to specify the file from which to read. The variant \BVerbatimInput puts the typeset result in a box without space above and below. The next example demonstrates some of the possibilities: it defines an environment example that first writes its body verbatim to a file, reads the first line back in and displays it in blue, reads the file once more, this time starting with the second line, and numbers the lines starting with the number 1. As explained above, a similar, albeit more complex definition was used to produce the examples in this book.

```
\usepackage{fancyvrb,color}
\newenvironment{example}
  {\VerbatimEnvironment\begin{VerbatimOut}{test.out}}
  {\end{VerbatimOut}\noindent
   \BVerbatimInput[lastline=1,formatcom=\color{blue}]{test.out}%
   \VerbatimInput[numbers=left,firstnumber=1,firstline=2]{test.out}}
\begin{example}
A blue line.
Two lines
with numbers.
\end{example}
```

```
   A blue line.

1  Two lines
2  with numbers.
```

3-4-22

An interesting set of sample environments can be found in the package fvrb-ex written by Denis Girou, which builds on the features provided by fancyvrb.

[1] In the first edition we unfortunately introduced a number of mistakes when showing code in text that was not directly used.

Variant environments and commands

So far, all examples have used the `Verbatim` environment, but there also exist a number of variants that are useful in certain circumstances. `BVerbatim` is similar to `Verbatim` but puts the verbatim lines into a box. Some keywords discussed above (notably those dealing with frames) are not supported, but two additional ones are available. The first, `baseline`, denotes the alignment point for the box; it can take the values `t` (for top), `c` (for center), or `b` (for bottom—the default). The second, `boxwidth`, specifies the desired width of the box; if it is missing or given the value `auto`, the box will be as wide as the widest line present in the environment. We already encountered `\BVerbatimInput`; it too, supports these additional keywords.

```
        first line
first line
        second line
second line
```

```
\usepackage{fancyvrb}
\begin{BVerbatim}[boxwidth=4pc,baseline=t]
first line
second line
\end{BVerbatim}
\begin{BVerbatim}[baseline=c]
first line
second line
\end{BVerbatim}
```

3-4-23

All environments and commands for typesetting verbatim text also have star variants, which, as in the standard LaTeX environments, display blanks as ␣. In other words, they internally set the keyword `showspaces` to `true`.

Defining your own variants

Defining customized variants of verbatim commands and environments is quite simple. For starters, the default settings built into the package can be changed with the help of the `\fvset` command. It takes one argument, a comma-separated list of key/value pairs. It applies them to every verbatim environment or command. Of course, you can still overwrite the new defaults with the optional argument on the command or environment. For example, if nearly all of your verbatim environments are indented by two spaces, you might want to remove them without having to deploy `gobble` on each occasion.

A line of text to show the left margin.

```
The new ‘normal’ case.

We now need to explicitly
cancel gobble occasionally!
```

```
\usepackage{fancyvrb}   \fvset{gobble=2}
\noindent A line of text to show the left margin.
\begin{Verbatim}
  The new ‘normal’ case.
\end{Verbatim}
\begin{Verbatim}[gobble=0]
We now need to explicitly
cancel gobble occasionally!
\end{Verbatim}
```

3-4-24

However, \fvset applies to all environments and commands, which may not be what you need. So the package offers commands to define your own verbatim environments and commands or to modify the behavior of the predefined ones.

```
\CustomVerbatimEnvironment     {new-env}{base-env}{key/val-list}
\RecustomVerbatimEnvironment{change-env}{base-env}{key/val-list}
\CustomVerbatimCommand         {new-cmd}{base-cmd}{key/val-list}
\RecustomVerbatimCommand    {change-cmd}{base-cmd}{key/val-list}
```

These declarations take three arguments: the name of the new environment or command being defined, the name of the environment or command (without a leading backslash) on which it is based, and a comma-separated list of key/value pairs that define the new behavior. To define new structures, you use \CustomVerbatimEnvironment or \CustomVerbatimCommand and to change the behavior of existing environments or commands (predefined ones as well as those defined by you), you use \RecustomVerbatimEnvironment or \RecustomVerbatimCommand. As shown in the following example, the default values, set in the third argument, can be overwritten as usual with the optional argument when the environment or command is instantiated.

```
1  The normal case with thick
2  rules and numbers on the left.

   The exception without numbers
   and thinner rules.

1  And␣from␣here␣on␣the␣environment
2  behaves␣differently␣again.
```

3-4-25

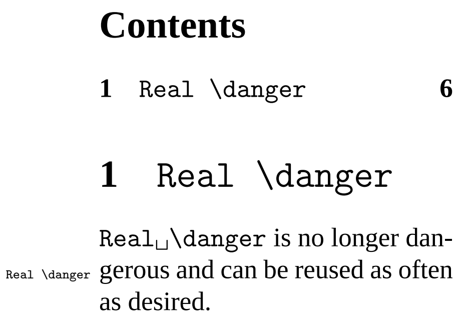
```
\usepackage{fancyvrb}
\CustomVerbatimEnvironment{myverbatim}{Verbatim}
        {numbers=left,frame=lines,framerule=2pt}
\begin{myverbatim}
The normal case with thick
rules and numbers on the left.
\end{myverbatim}
\begin{myverbatim}[numbers=none,framerule=.6pt]
The exception without numbers
and thinner rules.
\end{myverbatim}
\RecustomVerbatimEnvironment{myverbatim}{Verbatim}
         {numbers=left,frame=none,showspaces=true}
\begin{myverbatim}
And from here on the environment
behaves differently again.
\end{myverbatim}
```

Miscellaneous features

LaTeX's standard \verb command normally cannot be used inside arguments, because in such places the parsing mechanism would go astray, producing incorrect results or error messages. A solution to this problem is to process the verbatim data outside the argument, save it, and later use the already parsed data in such dangerous places. For this purpose the fancyvrb package offers the commands \SaveVerb and \UseVerb.

`\SaveVerb[`*key/val-list*`]{`*label*`}=` *data* `=`   `\UseVerb*[`*key/val-list*`]{`*label*`}`

The command `\SaveVerb` takes one mandatory argument, a *label* denoting the storage bin in which to save the parsed data. It is followed by the verbatim *data* surrounded by two identical characters (= in the syntax example above), in the same way that `\verb` delimits its argument. To use this data you call `\UseVerb` with the *label* as the mandatory argument. Because the data is only parsed but not typeset by `\SaveVerb`, it is possible to influence the typesetting by applying a list of key/value pairs or a star as with the other verbatim commands and environments. Clearly, only a subset of keywords make sense, irrelevant ones being silently ignored. The `\UseVerb` command is unnecessarily fragile, so you have to `\protect` it in moving arguments.



Contents

1 `Real \danger` 6

1 `Real \danger`

`Real␣\danger` is no longer dangerous and can be reused as often as desired.

`Real \danger`

```
\usepackage{fancyvrb}
\SaveVerb{danger}=Real \danger=

\tableofcontents

\section{\protect\UseVerb{danger}}

\UseVerb*{danger} is no longer dangerous
and can\marginpar{\UseVerb[fontsize=\tiny]
                            {danger}}
be reused as often as desired.
```

3-4-26

It is possible to reuse such a storage bin when it is no longer needed, but if you use `\UseVerb` inside commands that distribute their arguments over a large distance you have to be careful to ensure that the storage bin still contains the desired contents when the command finally typesets it. In the previous example we placed `\SaveVerb` into the preamble because the use of its storage bin inside the `\section` command eventually results in an execution of `\UseVerb` inside the `\tableofcontents` command.

`\SaveVerb` also accepts an optional argument in which you can put key/value pairs, though again only a few are relevant (e.g., those dealing with parsing). There is one additional keyword `aftersave`, which takes code to execute immediately after saving the verbatim text into the storage bin. The next example shows an application of this keyword: the definition of a special variant of the `\item` command that accepts verbatim text for display in a `description` environment. It also supports an optional argument in which you can put a key/value list to influence the formatting. The definition is worth studying, even though the amount of mixed braces and brackets seems distressingly complex at first. They are necessary to ensure that the right brackets are matched by `\SaveVerb`, `\item`, and `\UseVerb`—the usual problem, since brackets do not nest like braces do in TEX.[1] Also note the use of `\textnormal`, which is needed to cancel the `\bfseries` implicitly issued

[1] The author confesses that it took him three trials (close to midnight) to make this example work.

by the `\item` command. Otherwise, the `\emph` command in the example would not show any effect since no Computer Modern bold italic face exits.

`\ddanger` Dangerous beast; found in TEXbooks.

`\danger` Its small brother, still dangerous.

`\dddanger{`*arg*`}` The ultimate horror.

3-4-27

```
\usepackage{fancyvrb}
\newcommand\vitem[1][]{\SaveVerb[commandchars=\|\<\>,%
   aftersave={\item[\textnormal{\UseVerb[#1]{vsave}}]}]{vsave}}
\begin{description}
\vitem+\ddanger+   Dangerous beast;\\ found in \TeX books.
\vitem[fontsize=\tiny]+\danger+ Its small brother,
                                still dangerous.
\vitem+\dddanger{|emph<arg>}+   The ultimate horror.
\end{description}
```

In the same way you can save whole verbatim environments using the environment `SaveVerbatim`, which takes the name of a storage bin as the mandatory argument. To typeset them, `\UseVerbatim` or `\BUseVerbatim` (boxed version) with the usual key/value machinery can be used.

Even though verbatim commands or environments are normally not allowed inside footnotes, you do not need to deploy `\SaveVerb` and the like to get verbatim text into such places. Instead, place the command `\VerbatimFootnotes` at the beginning of your document (following the preamble!) and from that point onward, you can use verbatim commands directly in footnotes. However, this was only implemented for footnotes—for other commands, such as `\section`, you still need the more complicated storage bin method described above.

A bit of text to give us a reason to use a footnote.[1] Was this good enough?

[1] Here is proof: `\danger{%_^}`

3-4-28

```
\usepackage{fancyvrb}
\VerbatimFootnotes
A bit of text to give us a reason to use a
footnote.\footnote{Here is proof: \verb=\danger{%_^}=}
Was this good enough?
```

The fancyvrb version of `\verb` is called `\Verb`, and it supports all applicable keywords, which can be passed to it via an optional argument as usual. The example below creates `\verbx` as a variant of `\Verb` with a special setting of `commandchars` so that we can execute commands within its argument. We have to use `\CustomVerbatimCommand` for this purpose, since `\verbx` is a new command not available in standard LATEX.

`\realdanger{|emph<arg>}`
`\realdanger{`*arg*`}`

3-4-29

```
\usepackage{fancyvrb}
\CustomVerbatimCommand\verbx{Verb}{commandchars=\|\<\>}
\Verb[fontfamily=courier]+\realdanger{|emph<arg>}+ \\
\verbx[fontfamily=courier]+\realdanger{|emph<arg>}+
```

As already mentioned, fancyvrb offers a way to make a certain character denote the start and stop of verbatim text without the need to put `\verb` in front. The command to declare such a delimiting character is `\DefineShortVerb`.

Like other fancyvrb commands it accepts an optional argument that allows you to set key/value pairs. These influence the formatting and parsing, though this time you cannot overwrite your choices on the individual instance. Alternatively, `\fvset` can be used, since it works on all verbatim commands and environments within its scope. To remove the special meaning from a character declared with `\DefineShortVerb`, use `\UndefineShortVerb`.

The use of `\DefineShortVerb` can make sources much more readable—or unreadable!

And with `\UndefineShortVerb{\|}` we can return the | character back to normal.

```
\usepackage{fancyvrb}
\DefineShortVerb[fontsize=\tiny]{\|}
The use of |\DefineShortVerb| can make sources
much more readable---or unreadable! \par
\UndefineShortVerb{\|}\DefineShortVerb{\+}
\fvset{fontfamily=courier}
And with  +\UndefineShortVerb{\|}+
we can return the +|+ character back to normal.
```

3-4-30

Your favorite extensions or customizations can be grouped in a file with the name `fancyvrb.cfg`. After fancyvrb finishes loading, the package will automatically search for this file. The advantage of using such a file, when installed in a central place, is that you do not have to put your extensions into all your documents. The downside is that your documents will no longer be portable unless you distribute this file in tandem with them.

3.4.4 listings—Pretty-printing program code

A common application of verbatim typesetting is presenting program code. While on can successfully deploy a package like fancyvrb to handle this job, it is often preferable to enhance the display by typesetting certain program components (such as keywords, identifiers, and comments) in a special way.

Two major approaches are possible: one can provide commands to identify the logical aspects of algorithms or the programming language, or the application can (try to) analyze the program code behind the scenes. The advantage of the first approach is that you have potentially more control over the presentation; however, your program code is intermixed with TEX commands and thus may be difficult to maintain, unusable for direct processing, and often rather complicated to read in the source. Examples of packages classified into this category are alg and algorithmic. Here is an example:

```
if $i \leq 0$ then
  $i \leftarrow 1$
else
  if $i \geq 0$ then
    $i \leftarrow 0$
  end if
end if
```

```
\usepackage{algorithmic}
\begin{algorithmic}
\IF {$i\leq0$} \STATE $i\gets1$ \ELSE
\IF {$i\geq0$} \STATE $i\gets0$ \ENDIF
\ENDIF
\end{algorithmic}
```

3-4-31

ABAP (R/2 4.3, R/2 5.0, R/3 3.1, R/3 4.6C, R/3 6.10)
ACSL
Ada (83, 95)
Algol (60, 68)
Assembler (x86masm)
Awk (gnu, POSIX)
Basic (Visual)
C (ANSI, Objective, Sharp)
C++ (ANSI, GNU, ISO, Visual)
Caml (light, Objective)
Clean
Cobol (1974, 1985, ibm)
Comal 80
csh
Delphi
Eiffel
Elan
erlang
Euphoria
Fortran (77, 90, 95)
GCL
Gnuplot
Haskell
HTML
IDL (empty, CORBA)
Java (empty, AspectJ)
ksh
Lisp (empty, Auto)
Logo
Make (empty, gnu)
Mathematica (1.0, 3.0)
Matlab
Mercury
MetaPost
Miranda
Mizar
ML
Modula-2
MuPAD
NASTRAN
Oberon-2
OCL (decorative, OMG)
Octave
Pascal (Borland6, Standard, XSC)
Perl
PHP
PL/I
POV
Prolog
Python
R
Reduce
S (empty, PLUS)
SAS
Scilab
SHELXL
Simula (67, CII, DEC, IBM)
SQL
tcl (empty, tk)
TeX (AlLaTeX, common, LaTeX, plain, primitive)
VBScript
Verilog
VHDL (empty, AMS)
VRML (97)
XML

Table 3.7: Languages supported by listings (Winter 2003); blue indicates default dialect

The second approach is exemplified in the package listings[1] written by Carsten Heinz. This package first analyzes the code, decomposes it into its components, and then formats those components according to customizable rules. The package parser is quite general and can be tuned to recognize the syntax of many different languages (see Table 3.7). New languages are regularly added, so if your target language is not listed it might be worth checking the latest release of the package on CTAN. You may even consider contributing the necessary declarations yourself, which involves some work but is not very difficult.

The user commands and environments in this package share many similarities with those in fancyvrb. Aspects of parsing and formatting are controlled via key/value pairs specified in an optional argument, and settings for the whole document or larger parts of it can be specified using \lstset (the corresponding fancyvrb command is \fvset). Whenever appropriate, both packages use the same keywords so that users of one package should find it easy to make the transition to the other.

[1] The package version described here is 1.0. Earlier releases used a somewhat different syntax in some cases, so please upgrade if you find that certain features do not work as advertised.

After loading the package it is helpful to specify all program languages needed in the document (as a comma-separated list) using `\lstloadlanguages`. Such a declaration does not select a language, but merely loads the necessary support information and speeds up processing.

Program fragments are included inside a `lstlisting` environment. The language of the fragment is specified with the `language` keyword. In the following example we set this keyword via `\lstset` to `C` and then overwrite it later in the optional argument to the second `lstlisting` environment.

A “for” loop in C:

```
int sum;
int i; /* for loop variable */
sum=0;
for (i=0;i<n;i++) {
  sum += a[i];
}
```

Now the same loop in Ada:

```
Sum: Integer;
-- no decl for I necessary
Sum := 0;
for I in 1..N loop
  Sum := Sum + A(I);
end loop;
```

```
\usepackage{listings}
\lstloadlanguages{C,Ada}
\lstset{language=C,commentstyle=\scriptsize}
A ``for'' loop in C:
\begin{lstlisting}[keywordstyle=\underbar]
int sum;
int i; /*for loop variable*/
sum=0;
for (i=0;i<n;i++) {
  sum += a[i];
}
\end{lstlisting}
Now the same loop in Ada:
\begin{lstlisting}[language=Ada]
Sum: Integer;
-- no decl for I necessary
Sum := 0;
for I in 1..N loop
  Sum := Sum + A(I);
end loop;
\end{lstlisting}
```

3-4-32

This example also uses the keyword `commentstyle`, which controls the layout of comments in the language. The package properly identifies the different syntax styles for comments. Several other such keywords are available as well—`basicstyle` to set the overall appearance of the listing, `stringstyle` to format strings in the language, and `directivestyle` to format compiler directives, among others.

To format the language keywords, `keywordstyle` and `ndkeywordstyle` (second order) are used. Other identifiers are formatted according to the setting of `identifierstyle`. The values for the “style” keywords (except `basicstyle`) accept a one-argument LaTeX command such as `\textbf` as their last token. This scheme works because the “identifier text” is internally surrounded by braces and can thus be picked up by a command with an argument.

Thus, highlighting of keywords, identifiers, and other elements is done automatically in a customizable way. Nevertheless, you might want to additionally emphasize the use of a certain variable, function, or interface. For this purpose you can use the keywords emph and emphstyle. The first gets a list of names you want to emphasize; the second specifies how you want them typeset.

3-4-33

```
Sum: Integer;   Sum := 0;
for I in 1..N loop
  Sum := Sum + A(I);
end loop;
```

```
\usepackage{listings,color}
\lstset{emph={Sum,N},emphstyle=\color{blue},
        emph=[2]I,emphstyle=[2]\underbar}
\begin{lstlisting}[language=Ada]
Sum: Integer;   Sum := 0;
for I in 1..N loop
  Sum := Sum + A(I);
end loop;
\end{lstlisting}
```

If you want to typeset a code fragment within normal text you can use the command \lstinline. The code is delimited in the same way as with the \verb command, meaning that you can choose any character (other than the open bracket) that is not used within the code fragment and use it as delimiter. An open bracket cannot be used because the command also accepts an optional argument in which you can specify a list of key/value pairs.

3-4-34

The for loop is specified as i=0;i<n;i++.

```
\usepackage{listings}  \lstset{language=C}
The \lstinline[keywordstyle=\underbar]!for!
loop is specified as \lstinline!i=0;i<n;i++!.
```

Of course, it is also possible to format the contents of whole files; for this purpose you use the command \lstinputlisting. It takes an optional argument in which you can specify key/value pairs and a mandatory argument in which you specify the file name to process. In the following example, the package identifies keywords of case-insensitive languages, even if they are written in an unusual mixed-case (WrItE) manner.

3-4-35

```
for i:=1 to maxint do
begin
  WrItE('This␣is␣stupid');
end.
```

```
\usepackage{listings}
\begin{filecontents*}{pascal.src}
for i:=1 to maxint do
begin
  WrItE('This is stupid');
end.
\end{filecontents*}
\lstinputlisting[language=Pascal]{pascal.src}
```

Spaces in strings are shown as ␣ by default. This behavior can be turned off by setting the keyword showstringspaces to false, as seen in the next example. It is also possible to request that all spaces be displayed in this way by setting the keyword showspaces to true. Similarly, tab characters can be made visible by using the Boolean keyword showtabs.

Line numbering is possible, too, using the same keywords as employed with fancyvrb: `numbers` accepts either `left`, `right`, or `none` (which turns numbering on or off), `numberblanklines` decides whether blank lines count with respect to numbering (default `false`), `numberstyle` defines the overall look and feel of the numbers, `stepnumber` defines which line numbers will appear (`0` means no numbering), and `numbersep` defines the separation between numbers and the start of the line. By default, line numbering starts with 1 on each `\lstinputlisting` but this can be changed using the `firstnumber` keyword. If you specify `last` as a special value to `firstnumber`, numbering is continued.

Some text before ...

```
10 for i:=1 to maxint do
   begin
12   WrItE('This is stupid');
   end.
```

```latex
\usepackage{listings}
% pascal.src as defined before
\lstset{numberstyle=\tiny,numbers=left,
   stepnumber=2,numbersep=5pt,firstnumber=10,
   xleftmargin=12pt,showstringspaces=false}
\noindent Some text before \ldots
\lstinputlisting[language=Pascal]{pascal.src}
```

3-4-36

An overall indentation can be set using the `xleftmargin` keyword, as shown in the previous example, and `gobble` can be used to remove a certain number of characters (hopefully only spaces) from the left of each line displayed. Normally, indentations of surrounding environments like `itemize` will be honored. This feature can be turned off using the Boolean keyword `resetmargin`. Of course, all such keywords can be used together. To format only a subrange of the code lines you can specify the first and/or last line via `firstline` and `lastline`; for example, `lastline=10` would typeset a maximum of 10 code lines.

Another way to provide continued numbering is via the `name` keyword. If you define "named" environments using this keyword, numbering is automatically continued with respect to the previous environment with the same name. This allows independent numbering if the need arises.

```
Sum: Integer;                 1
```

The second fragment continues the numbering.

```
Sum := 0;                     2
for I in 1..N loop            3
   Sum := Sum + A(I);         4
end loop;                     5
```

```latex
\usepackage{listings} \lstset{language=Ada,numbers=right,
   numberstyle=\tiny,stepnumber=1,numbersep=5pt}
\begin{lstlisting}[name=Test]
Sum: Integer;
\end{lstlisting}
The second fragment continues the numbering.
\begin{lstlisting}[name=Test]
Sum := 0;
for I in 1..N loop
  Sum := Sum + A(I);
end loop;
\end{lstlisting}
```

3-4-37

If a listing contains very long lines they may not fit into the available measure. In that case listings will produce overfull lines sticking out to the right, just like a verbatim environment would do. However, you can direct it to break long lines at spaces or punctuation characters by specifying the keyword breaklines. Wrapped lines are indented by 20pt, a value that can be adjusted through the keyword breakindent.

If desired, you can add something before (keyword prebreak) and after (keyword postbreak) the break to indicate that the line was artifically broken in the listing. We used this ability below to experiment with small arrows and later on with the string "(cont.)" in tiny letters. Both keywords are internally implemented as a TEX \discretionary, which means that they accept only certain input (characters, boxes, and kerns). For more complicated material it would be best to wrap everything in an \mbox, as we did in the example. In case of color changes, even that is not enough: you need an extra level of braces to prevent the color \special from escaping from the box (see the discussion in Appendix A.2.5).

The example exhibits another feature of the breaking mechanism—namely, if spaces or tabs appear in front of the material being broken, then these spaces are by default repeated on continuation lines. If this behavior is not desired, set the keyword breakautoindent to false as we did in the second part of the example.

```
Text at left margin
        /*A long ↘
        →string is ↘
        →broken ↘
        →across the ↘
        →line!*/

        /*A long ↘
(cont.) string is broken ↘
(cont.) across the line!*/
```

3-4-38

```
\usepackage{color,listings}
\lstset{breaklines=true,breakindent=0pt,
        prebreak=\mbox{\tiny$\searrow$},
        postbreak=\mbox{{\color{blue}\tiny$\rightarrow$}}}
\begin{lstlisting}
Text at left margin
        /*A long string is broken across the line!*/
\end{lstlisting}
\begin{lstlisting}[breakautoindent=false,
                   postbreak=\tiny (cont.)\,]
        /*A long string is broken across the line!*/
\end{lstlisting}
```

You can put frames or rules around listings using the frame keyword, which takes the same values as it does in fancyvrb (e.g., single, lines). In addition, it accepts a subset of the string trblTRBL as its value. The uppercase letters stand for double rules the lowercase ones for single rules. There are half a dozen more keywords: to influence rule widths, create separation from the text, make round corners, and so on—all of them are compatible with fancyvrb if the same functionality is provided.

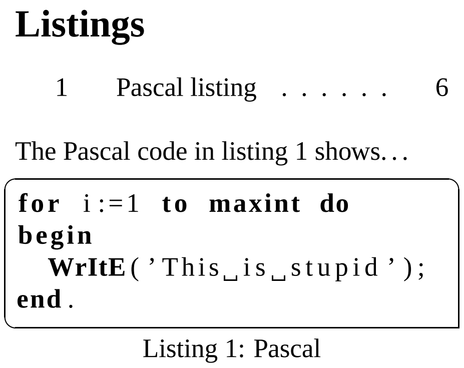
```
for␣i:=1␣to␣maxint␣do
begin
␣␣WrItE('This␣is␣stupid');
end.
```

3-4-39

```
\usepackage{listings}
% pascal.src as defined before
\lstset{frame=trBL,framerule=2pt,framesep=4pt,
        rulesep=1pt,showspaces=true}
\lstinputlisting[language=Pascal]{pascal.src}
```

You can specify a caption for individual listings using the keyword caption. The captions are, by default, numbered and prefixed with the string Listing stored in \lstlistingname. The counter used is lstlisting; thus, to change its appearance you could modify \thelstlisting. The caption is positioned either above (default) or below the listing, and this choice can be adjusted using the keyword captionpos.

To get a list of all captions, put the command \lstlistoflistings at an appropriate place in your document. It produces a heading containing the words stored in \lstlistlistingname (default is Listings). If you want the caption text in the document to differ from the caption text in the list of listings, use an optional argument as shown in the following example. Note that in this case you need braces around the value to hide the right bracket. To prevent the caption from appearing in the list of listings, use the keyword nolol with a value of true. By using the keyword label you can specify a label for referencing the listing number via \ref, provided you have not suppressed the number.



Listings

1 Pascal listing 6

The Pascal code in listing 1 shows. . .

```
for i:=1 to maxint do
begin
  WrItE('This␣is␣stupid');
end.
```

Listing 1: Pascal

```
\usepackage{listings}
% pascal.src as defined before
\lstset{frame=single,frameround=tftt,
        language=Pascal,captionpos=b}
\lstlistoflistings
          %
\bigskip  % normally the above is in the
\noindent % front matter section, but here ...
          %
The Pascal code in listing~\ref{foo} shows\ldots
\lstinputlisting
   [caption={[Pascal listing]Pascal},label=foo]
   {pascal.src}
```

3-4-40

The keyword frameround used in the previous example allows you to specify round corners by giving t for true and f for false, starting with the upper-right corner and moving clockwise. This feature is not available with fancyvrb frames.

Instead of formatting your listings within the text, you can turn them into floats by using the keyword float, typically together with the caption keyword. Its value is a subset of htbp specifying where the float is allowed to go (using it without a value is equivalent to tbp). You should, however, avoid mixing floating and nonfloating listings as this could sometimes result in captions being numbered out of order, as in Example 6-3-5 on page 296.

By default, listings only deals with input characters in the ASCII range; unexpected 8-bit input can produce very strange results, like the misordered letters in the following example. By setting extendedchars to true you can enable the use of 8-bit characters, which makes the package work harder, but (usually) produces the right results. Of course, if you use an extended character set you would normally add the keyword to the \lstset declaration instead of specifying it every time on the environment. It is also possible to specify an input encoding for the code fragments (if different from the input encoding used for the remainder of the document) by using the keyword inputencoding. This keyword can be used only if the inputenclistings package is loaded.

```
int i; /*üfr die äßuere Schleife*/

int i; /*für die äußere Schleife*/
```

3-4-41

```
\usepackage[latin1]{inputenc}
\usepackage{listings}
\lstset{language=C,commentstyle=\scriptsize}
\begin{lstlisting}
int i; /*für die äußere Schleife*/
\end{lstlisting}
\begin{lstlisting}[extendedchars=true]
int i; /*für die äußere Schleife*/
\end{lstlisting}
```

The package offers many more keys to influence the presentation. For instance, you can escape to LaTeX for special formatting tricks, display tab or form-feed characters, index certain identifiers, or interface to hyperref so that clicking on some identifier will jump to the previous occurrence. Some of the features are still considered experimental and you have to request them using an optional argument during package loading. These are all documented in great detail in the manual (roughly 50 pages) accompanying the package.

As a final example of the kind of treasures you can find in that manual, look at the following example. It shows code typesetting as known from Donald Knuth's literate programming conventions.

```
var i:integer;
if (i≤0) i ← 1;
if (i≥0) i ← 0;
if (i≠0) i ← 0;
```

3-4-42

```
\usepackage{listings}
\lstset{literate={:=}{{$\gets$}}1
 {<=}{{$\leq$}}1 {>=}{{$\geq$}}1 {<>}{{$\neq$}}1}
\begin{lstlisting}[gobble=2]
  var i:integer;
  if (i<=0) i := 1;
  if (i>=0) i := 0;
  if (i<>0) i := 0;
\end{lstlisting}
```

3.5 Lines and columns

In the last part of this chapter we present a few packages that help in manipulating the text stream in its entirety. The first package deals with attaching line numbers to paragraphs, supporting automatic references to them. This can be useful in critical editions and other scholarly works.

The second package deals with the problem of presenting two text streams side by side—for example, some original and its translation. We will show how both packages can be combined in standard cases.

The third package deals with layouts having multiple columns. It allows switching between different numbers of columns on the same page and supports balancing textual data. Standard LaTeX already offers the possibility of typesetting text in one- or two-column mode, but one- and two-column output cannot be mixed on the same page.

We conclude by introducing a package that allows you to mark the modifications in your source with vertical bars in the margin.

3.5.1 lineno—Numbering lines of text

In certain applications it is useful or even necessary to number the lines of paragraphs to be able to refer to them. As TeX optimizes the line breaking over the whole paragraph, it is ill equipped to provide such a facility, since technically line breaking happens at a very late stage during the processing, just before the final pages are constructed. At that point macro processing, which could add the right line number or handle automatic references, has already taken place. Hence, the only way to achieve line numbering is by deconstructing the completed page line by line in the "output routine" (i.e., the part of LaTeX, that normally breaks the paragraph galley into pages and adds running headers and footers) and attaching the appropriate line numbers at that stage.

This approach was taken by Stephan Böttcher in his lineno package. Although one would expect such an undertaking to work only in a restricted environment, his package is surprisingly robust and works seamlessly with many other packages—even those that modify the LaTeX output routine, such as ftnright, multicol, and wrapfig. It also supports layouts produced with the `twocolumn` option of the standard LaTeX classes.

```
\linenumbers*[start-number]    \nolinenumbers
```

Loading the lineno package has no direct effect: to activate line numbering, a `\linenumbers` command must be specified in the preamble or at some point in the document. The command `\nolinenumbers` deactivates line numbering again. Line numbering works on a per-paragraph basis. Thus, when LaTeX sees the end of a paragraph, it checks whether line numbering is currently requested and, if so, attaches numbers to *all* lines of that paragraph. It is therefore best to put these commands between paragraphs rather than within them.

The `\linenumbers` command can take an optional argument that denotes the number to use for the first line. If used without such an argument, it continues from where it stopped numbering previously. You can also use a star form, which is a shorthand for `\linenumbers[1]`.

No line numbers here. Some text to experiment with line numbering.

1 But here we get line numbers. Some
2 text to experiment with line numbering.
3 And here too. Some text to experiment
4 with line numbering.
-10 Restart with a negative number. Some
-9 text to experiment with line numbering.

3-5-1

```
\usepackage{lineno}
\newcommand\para{ Some text to experiment
                  with line numbering.\par}
No line numbers here.\para
\linenumbers
But here we get line numbers. \para
And here too. \para
\linenumbers[-10]
Restart with a negative number. \para
```

Rather than starting or stopping line numbering with the above commands, you can use the environment `linenumbers` to define the region that should get line numbers. This environment will automatically issue a `\par` command at the end to terminate the current paragraph. If line numbers are needed only for short passages, the environment form (or one of the special environments `numquote` and `numquotation` described later) is preferable.

Numbering boxed paragraphs

As the production of line numbers involves the output routine, numbering will take place only for paragraphs being built and put on the "main vertical list" but not for those built inside boxes (e.g., not inside a `\marginpar` or within the body of a float). However, the package offers some limited support for numbering lines in such places via the `\internallinenumbers` command. Restrictions are that the baselines within such paragraphs need to be a fixed distance apart (otherwise, the numbers will not get positioned correctly) and that you may have to end such paragraphs with explicit `\par` commands. The `\internallinenumbers` command accepts a star and an optional argument just as `\linenumbers` does. However, the starred form not only ensures that line numbering is (re)started with 1, but also that the line numbers do not affect line numbering in the main vertical list; compare the results in the two `\marginpar`s below.

1 Some text on the main verti-
2 cal list! Some text to experiment
3 with line numbering.
4 Some text to experiment with
5 line numbering.
9 In this paragraph we use
10 a second marginal note affect-
11 ing the line numbers this time.
12 Some text to experiment with
13 line numbering.

1 Some text to experi-
2 ment with line num-
3 bering.

6 Some text to experi-
7 ment with line num-
8 bering.

3-5-2

```
\usepackage{lineno}
% \para defined as before
\linenumbers
Some text on the main vertical list!
\marginpar{\footnotesize
           \internallinenumbers* \para}
\para \para In this paragraph we use
a second marginal note affecting the
\marginpar{\footnotesize
           \internallinenumbers  \para}
line numbers this time. \para
```

The line numbers in the second `\marginpar` continue the numbering on the main vertical list (the last line of first paragraph was 5) and the second paragraph then continues with line number 9. Such \marginpar commands are processed before the paragraph containing them is broken into lines, which explains the ordering of the numbers.

Handling display math

As lineno needs \par to attach line numbers when the output routine is invoked, a TEXnical problem arises when certain display math constructs are used: the partial paragraph above such a display is broken into lines by TEX without issuing a \par. As a consequence, without further help such a partial paragraph will not get any line numbers attached. The package's solution, as illustrated in the next example, is to offer the environment linenomath, which, if it surrounds such a display, will take care of the line numbering problem. It also has a starred form that also numbers the display lines.

No line number before the display:

$$x \neq y$$

1 Some text to experiment with line numbering.

2 But line numbers in this case:

$$x \neq y$$

3 Some text to experiment with line numbering.

```
\usepackage{lineno} \linenumbers
\newcommand\sample{ Some text to
      experiment with line numbering.}
No line number before the display:
\[ x \neq y \] \sample \par
But line numbers in this case:
\begin{linenomath}
   \[ x \neq y \]
\end{linenomath}
\sample\par
```

3-5-3

If there are many such displays the need for surrounding each of them with a linenomath environment is cumbersome. For this reason the package offers the option displaymath, which redefines the basic LATEX math display environments so that they internally use linenomath environments. The option mathlines will make linenomath behave like its starred form so that the displayed mathematical formulas get line numbers as well.

1 Some text to experiment with line numbering.

2 $$x \neq y$$

3 Some text to experiment with line numbering.

4 Some text to experiment with line numbering.

5 $$x \neq y$$

6 Some text to experiment with line numbering.

```
\usepackage[displaymath,mathlines]
                     {lineno}
\linenumbers
% \sample as defined before
\sample \[ x \neq y \] \sample\par
\sample
\begin{displaymath}
   x \neq y
\end{displaymath}
\sample
```

3-5-4

Cross-references to line numbers

To reference line numbers put a \linelabel into the line and then refer to it via \ref or \pageref, just as with other references defined using \label. The exception is that \linelabel can only be used on the main vertical list and should only be used within paragraphs that actually carry numbers. If it is used elsewhere, you get either a bogus reference (if the current line does not have a line number) or an error message (in places where \linelabel is not allowed).

1 Some text to experiment with line num-
2 bering. Some text to experiment with line
3 numbering. Some text to experiment with
4 line numbering. Some text to experiment
5 with line numbering. Some text to exper-
6 iment with line numbering.
7 In the text on lines 2, 3, up to and includ-
8 ing line 5 we see to refer to individual lines
9 ...

3-5-5

```
\usepackage{lineno}
\linenumbers
% \sample as defined before

\sample\linelabel{first} \sample \sample
\sample\linelabel{second} \sample

In the text on lines~\ref{first},
\lineref[1]{first}, up to and including
line~\ref{second} we see to refer to
individual lines \ldots
```

It is also possible to refer to a line that carries no \linelabel, by using the \lineref command with an optional argument specifying the offset. This ability can be useful if you need to refer to a line that cannot be easily labeled, such as a math display, or if you wish to refer to a sequence of lines, as in the previous example.

Labeling only some lines

There are several ways to customize the visual appearance of line numbers. Specifying the option modulo means that line numbers will only appear on some lines (default is every fifth). This effect can also be achieved by using the command \modulolinenumbers. Calling this command with an optional argument attaches numbers to lines that are multiples of the specified number (in particular, a value of 1 corresponds to normal numbering). Neither command nor option initiates line numbering mode, for that a \linenumbers command is still necessary.

1 Some text to experiment with line num-
2 bering. Some text to experiment with line
3 numbering. Some text to experiment with
4 line numbering.
And now a paragraph with numbers on
6 every second line. Some text to experiment
with line numbering. Some text to experi-
8 ment with line numbering. Some text to ex-
periment with line numbering.

3-5-6

```
\usepackage{lineno}
\linenumbers
% \sample defined as before

\sample \sample \sample \par
\modulolinenumbers[2]
And now a paragraph with numbers on every
second line. \sample \sample \sample \par
```

The font for line numbers is controlled by the hook \linenumberfont. Its default definition is to use tiny sans serif digits. The numbers are put flush right in a box of width \linenumberwidth. This box is separated from the line by the value stored in \linenumbersep. To set the number flush left you have to dig deeper, but even for this case you will find hooks like \makeLineNumberRight in the package. Although changing the settings in the middle of a document is usually not a good idea, it was done in the next example for demonstration purposes.

The option “right” changes the line num- 1
ber position. Some text to experiment with 2
line numbering. Some text to experiment 3
with line numbering. 4
Now we use a different font and a big- 5
ger separation. Some text to experiment 6
with line numbering. Some text to experi- 7
ment with line numbering. 8

```
\usepackage[right]{lineno}
\linenumbers
% \sample defined as before
The option ‘‘right’’ changes the line
number position. \sample \sample \par
\renewcommand\linenumberfont
  {\normalfont\footnotesize\ttfamily}
\setlength\linenumbersep{20pt}
Now we use a different font and a bigger
separation. \sample \sample \par
```

3-5-7

For special applications the package offers two environments that provide line numbers automatically: `numquote` and `numquotation`. They are like their LaTeX cousins `quote` and `quotation`, except that their lines are numbered. They accept an optional argument denoting the line number with which to start (if the argument is omitted, they restart with 1) and they have starred forms that will suppress reseting the line numbers.

The main difference from their LaTeX counterparts (when used together with the `\linenumbers` command) is the positioning of the numbers, which are indented inward. Thus, their intended use is for cases when only the quoted text should receive line numbers that can be referenced separately.

1 Some text to experiment with line
2 numbering.

3 Some text to experiment with line number-
4 ing. Some text to experiment with line num-
5 bering.

1 Some text to experiment with line
2 numbering.

3 Some more text.

```
\usepackage{lineno}
\linenumbers
% \sample defined as before
\begin{quote}
 \sample
\end{quote}
\sample \sample
\begin{numquote}
  \sample
\end{numquote}
Some more text.
```

3-5-8

Providing your own extensions

Using the machinery provided by the package material, it is fairly easy to develop your own environments that attach special items to each line. The main macro to customize is `\makeLineNumber`, which gets executed inside a box of zero width at the left edge of each line (when line numbering mode is turned on). The net effect of your code should take up no space, so it is best to operate with `\llap` or `\rlap`. Apart from that you can use basically anything. You should only remember that the material is processed and attached after the paragraph has been broken into lines and normal macro-processing has finished, so, you should not expect it to interact with data in mid-paragraph. You can produce the current line number with the `\LineNumber` command, which will supply the number or nothing, depending on whether line numbering mode is on.

The following example shows the definition and use of two new environments that (albeit somewhat crudely, as they do not care about setting fonts and the like) demonstrate some of the possibilities. Note that even though the second environment does not print any line numbers, the lines are internally counted, so that line numbering resumes afterwards with the correct value.

```
 1→     Some  text  to  experiment
 2→ with line numbering.
        Some  text  to  experiment ←
    with line numbering.  Some text ←
    to experiment with line number- ←
    ing.                            ←
 7→     Some  text  to  experiment
 8→ with line numbering.  Some text
 9→ to experiment with line number-
10→ ing.
```

3-5-9

```
\usepackage{lineno} \linenumbers
% \sample defined as before
\newenvironment{numarrows}
    {\renewcommand\makeLineNumber
                  {\llap{\LineNumber$\rightarrow$ }}}
    {\par}
\newenvironment{arrows}{\renewcommand\makeLineNumber
   {\rlap{\hspace{\textwidth} $\leftarrow$}}}{\par}
\begin{numarrows}  \sample  \end{numarrows}
\begin{arrows} \sample \sample \end{arrows}
\sample
\begin{numarrows}  \sample  \end{numarrows}
```

The appearance and behavior of the line numbers can be further controlled by a set of options or, alternatively, by a set of commands equivalent to the options (see the package documentation for details on the command forms). With the options `left` (default) and `right`, you specify in which margin the line numbers should appear. Using the option `switch` or `switch*`, you get them in the outer and inner margins, respectively.

At least two LaTeX runs of the document are required before the line numbers will appear in the appropriate place. Unfortunately, there is no warning about the need to rerun the document, so you have to watch out for this issue yourself.

You can also request that numbers restart on each page by specifying the option `pagewise`. This option needs to come last.

3.5.2 parallel—Two text streams aligned

Sometimes it is necessary to typeset something in parallel columns, such as when presenting some text and its translation. Parallel in this context means that at certain synchronization points the two text streams are vertically (re)aligned. This type of layout is normally not supported by LaTeX (which by default only works with a single text stream), but it can be achieved by using Matthias Eckermann's parallel package.

This package provides the `Parallel` environment, which surrounds the material to be typeset in parallel. It takes two mandatory arguments: the widths of the left and right columns. Their sum should be less than `\textwidth`; otherwise, the text in the two columns will touch or even overlap. To ease usage, one or both arguments can be left empty, in which case the appropriate width for the column(s) will be calculated automatically, using the current value of `\ParallelUserMidSkip` as the column separation. To mark up the left and the right text streams, you use

\verb *is allowed*

\ParallelLText and \ParallelRText, respectively. Although both commands expect the text as an argument, it is nevertheless possible to use \verb or a verbatim environment inside, as the following example shows.

This is text in the English language explaining the command \foo.	Dies ist Text in deutscher Sprache, der das Kommando \foo erläutert.

```
\usepackage{parallel}
\begin{Parallel}{}{}
 \ParallelLText{This is text in the English
   language explaining the command \verb=\foo=.}
 \ParallelRText{Dies ist Text in deutscher Sprache,
   der das Kommando \verb=\foo= erl\"autert.}
\end{Parallel}
```

3-5-10

To align certain lines of text you split the two text streams at appropriate points by using pairs of \ParallelLText and \ParallelRText commands and separating each pair with \ParallelPar. If you forget one of the \ParallelPar commands, some of your text will get lost without warning. Moreover, as its name suggests, the \ParallelPar command introduces a paragraph break, so that alignment is possible only at paragraph boundaries. Additional paragraph breaks inside the argument of an \Parallel..Text command are also possible but in that case no alignment is attempted.

In the next example, displaying a few "direct" translations of computer lingua into German (taken from [54] with kind permission by Eichborn Verlag), we define a shorthand command \LR to make it easier to input the text. If such a shorthand is used, \verb can no longer be used in the argument. Thus, if you need \verb, use the package commands directly. We also use the lineno package since line numbers can be useful when talking about a text and its translation.

I just go online and download an update.	Ich geh mal eben auf den Strich und lade mir ein Auffrisch herunter.
This laptop is missing several interfaces.	Dieser Schoßspitze fehlt so manches Zwischengesicht.
Microsoft Office on floppy disks.	Kleinweich Büro auf Schlabberscheiben.

```
\usepackage{parallel,lineno}
\linenumbers \modulolinenumbers[2]
\setlength\linenumbersep{1pt}
\newcommand\LR[2]{\ParallelLText{#1}%
                  \ParallelRText{#2}\ParallelPar}
\begin{Parallel}{.45\linewidth}{}
\raggedright   \setlength\leftskip{10pt}
               \setlength\parindent{-10pt}
\LR{I just go online and download an update.}{Ich
  geh mal eben auf den Strich und lade mir ein
  Auffrisch herunter.}  \LR{This laptop is missing
 several interfaces.} {Dieser Scho\ss\-spitze
 fehlt so manches Zwi\-schen\-ge\-sicht.}
\LR{Microsoft Office on floppy disks.}{Kleinweich
  B\"uro auf Schlabberscheiben.}
\end{Parallel}
```

3-5-11

As you can see, it is possible to adjust paragraph parameters within the scope of the Parallel environment. The negative \parindent cancels the positive `\leftskip` so that each paragraph starts flush left but following lines are indented by `\leftskip` (and both must be changed *after* calling `\raggedright`, as the latter also sets these registers).

The `Parallel` environment works by aligning line by line, which has a surprising consequence when one block contains unusually large objects, such as a display. Thus, the method is suitable only for normal text lines.

This is text that contains:

$$\sum_{n=1}^{x} 2a_n$$

And here is the explanation showing some surprising effect.

3-5-12

```
\usepackage{parallel}
\begin{Parallel}{}{}
 \ParallelLText{This is text that contains:
                  \[ \sum_{n=1}^x2 a_n \]}
 \ParallelRText{And here is the explanation
                 showing some surprising effect.}
\end{Parallel}
```

Footnotes in parallel text

Footnotes within the parallel text are not placed at the bottom of the current page, but rather are typeset directly after the end of the current `Parallel` environment and separated from it by the result of executing `\ParallelAtEnd`, which is a command defined to do nothing. You can, however, redefine it to place something between footnotes and preceding text. If the redefinition should apply only to a single `Parallel` environment, place it within the scope of the environment.

The presentation of the footnotes is controlled by four package options: `OldStyleNums` sets footnote numbers using old-style numerals, `RaiseNums` generates raised footnote numbers, and `ItalicNums` produces italic numbers. If none of these options is given, then Arabic numerals at the baseline position are used. The options affect only the numbers in front of the footnote text; the markers within the parallel text are always raised Arabic numerals. The fourth option, `SeparatedFootnotes`, can be combined with one of the three other options and indicates that footnotes in each column should be independently numbered. The numbers from the right column are then postfixed with `\ParallelDot`, which by default produces a centered dot. In the next example its definition is slightly modified so that the dot itself does not take up any space.

This is text in the English language[1] explaining the command `\foo`.

Dies ist Text[1] in deutscher Sprache[2], der das Kommando `\foo` erläutert.

1 We hope!

1· Ein Satz.

2· Schlechter Stil!

3-5-13

```
\usepackage[OldStyleNums,SeparatedFootnotes]{parallel}
\renewcommand\ParallelAtEnd{\vspace{7pt}\footnoterule}
\renewcommand\ParallelDot
             {\makebox[0pt][l]{\textperiodcentered}}
\begin{Parallel}[v]{}{} \raggedright
 \ParallelLText{This is text in the English
   language\footnote{We hope!} explaining the
   command \verb=\foo=.}
 \ParallelRText{Dies ist Text\footnote{Ein Satz.} in
   deutscher Sprache\footnote{Schlechter Stil!}, der
   das Kommando \verb=\foo= erl\"autert.}
\end{Parallel}
```

The `Parallel` environment can sport an optional argument before the mandatory ones, whose value can be `c` (make two columns—the default), `v` (separate columns with a vertical rule as shown in the previous example), or `p` (put left text on left-hand pages and right text on right-hand pages). If the "page" variant is chosen it is possible that you get empty pages. For example, if you are on a verso page the environment has to skip to the next recto page in order to display the texts on facing pages.

3.5.3 multicol—A flexible way to handle multiple columns

With standard LaTeX it is possible to produce documents with one or two columns (using the class option `twocolumn`). However, it is impossible to produce only parts of a page in two-column format as the commands `\twocolumn` and `\onecolumn` always start a fresh page. Additionally, the columns are never balanced, which sometimes results in a slightly weird distribution of the material.

The multicol package[1] by Frank Mittelbach solves these problems by defining an environment, `multicols`, with the following properties:

- Support is provided for 2–10 columns, which can run for several pages.
- When the environment ends, the columns on the last page are balanced so that they are all of nearly equal length.
- The environment can be used inside other environments, such as `figure` or `minipage`, where it will produce a box containing the text distributed into the requested number of columns. Thus, you no longer need to hand-format your layout in such cases.
- Between individual columns, vertical rules of user-defined widths can be inserted.
- The formatting can be customized globally or for individual environments.

```
\begin{multicols}{columns}[preface][skip]
```

Normally, you can start the environment simply by specifying the number of desired columns. By default paragraphs will be justified, but with narrow measures—as in the examples—they would be better set unjustified as we show later on.

Here is some text to be distributed over several columns. If the columns are very narrow try typesetting ragged right.

```
\usepackage{multicol}
\begin{multicols}{3}
 Here is some text to be distributed over
 several columns. If the columns are very
 narrow try typesetting ragged right.
\end{multicols}
```

3-5-14

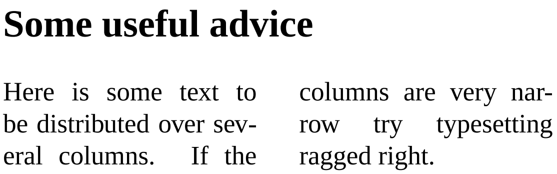
[1] For historical reasons the copyright of the multicol package, though distributed under LPPL (LaTeX Project Public License) [111], contains an additional "moral obligation" clause that asks commercial users to consider paying a license fee to the author or the LaTeX3 fund for their use of the package. For details see the head of the package file itself.

`\premulticols`	`50.0pt`	`\postmulticols`	`20.0pt`
`\columnsep`	`10.0pt`	`\columnseprule`	`0.0pt`
`\multicolsep`	`12.0pt plus 4.0pt minus 3.0pt`		

Table 3.8: Length parameters used by `multicols`

You may be interested in prefixing the multicolumn text with a bit of single-column material. This can be achieved by using the optional *preface* argument. LaTeX will then try to keep the text from this argument and the start of the multicolumn text on the same page.

3-5-15



Some useful advice

Here is some text to be distributed over several columns. If the columns are very narrow try typesetting ragged right.

```
\usepackage{multicol}
\begin{multicols}{2}
            [\section*{Some useful advice}]
 Here is some text to be distributed over
 several columns. If the columns are very
 narrow try typesetting ragged right.
\end{multicols}
```

The `multicols` environment starts a new page if there is not enough free space left on the current page. The amount of free space is controlled by a global parameter. However, when using the optional argument the default setting for this parameter may be too small. In this case you can either change the *global* default (see below) or adjust the value for the *current* environment by using a second optional *skip* argument as follows:

```
\begin{multicols}{3}[\section*{Index}][7cm]
  Text Text Text Text ...
\end{multicols}
```

This would start a new page if less than 7 cm free vertical space was available.

Preventing balancing

The `multicols` environment balances the columns on the last page (it was originally developed for exactly this purpose). If this effect is not desired you can use the `multicols*` variant instead. Of course, this environment works only in the main vertical galley, since inside a box one has to balance the columns to determine a column height.

The `multicols` environment recognizes several formatting parameters. Their meanings are described in the following sections. The default values can be found in Table 3.8 (dimensions) and Table 3.9 (counters). If not stated otherwise, all changes to the parameters have to be placed before the start of the environment to which they should apply.

The required free space

The `multicols` environment first checks whether the amount of free space left on the page is at least equal to `\premulticols` or to the value of the second optional argument, when specified. If the requested space is not available, a

\multicolpretolerance	-1	\multicoltolerance	9999
columnbadness	10000	finalcolumnbadness	9999
collectmore	0	unbalance	0
tracingmulticols	0		

Table 3.9: Counters used by multicols

\newpage is issued. A new page is also started at the end of the environment if the remaining space is less than \postmulticols. Before and after the environment, a vertical space of length \multicolsep is placed.

Column width and separation

The column width inside the multicols environment will automatically be calculated based on the number of requested columns and the current value of \linewidth. It will then be stored in \columnwidth. Between columns a space of \columnsep is left.

Adding vertical lines

Between any two columns, a rule of width \columnseprule is placed. If this parameter is set to 0pt (the default), the rule is suppressed. If you choose a rule width larger than the column separation, the rule will overprint the column text.

```
\usepackage{multicol,ragged2e}
\setlength\columnseprule{0.4pt}
\addtolength\columnsep{2pt}
\begin{multicols}{3}
\RaggedRight
  Here is some text to be distributed over
  several columns. In this example ragged-right
  typesetting is used.
\end{multicols}
```

Here is some text to be distributed over several columns. In this example ragged-right typesetting is used.

3-5-16

Column formatting

By default (the \flushcolumns setting), the multicols environment tries to typeset all columns with the same length by stretching the available vertical space inside the columns. If you specify \raggedcolumns the surplus space will instead be placed at the bottom of each column.

Paragraphs are formatted using the default parameter settings (as described in Sections 3.1.11 and 3.1.12) with the exception of \pretolerance and \tolerance, for which the current values of \multicolpretolerance and \multicoltolerance are used, respectively. The defaults are -1 and 9999, so that the paragraph-breaking trial without hyphenation is skipped and relatively bad paragraphs are allowed (accounting for the fact that the columns are typically very narrow). If the columns are wide enough, you might wish to change these defaults to something more restrictive, such as

```
    \multicoltolerance=3000
```

Note the somewhat uncommon assignment form: \multicoltolerance is an internal TEX counter and is controlled in exactly the same way as \tolerance.

Balancing control

At the end of the multicols environment, remaining text will be balanced to produce columns of roughly equal length. If you wish to place more text in the left columns you can advance the counter unbalance. This counter determines the number of additional lines in the columns in comparison to the number that the balancing routine has calculated. It will automatically be restored to zero after the environment has finished. To demonstrate the effect, the next example uses the text from Example 3-5-16 on the facing page but requests one extra line.[1]

3-5-17

Here is some text to be distributed over several columns. In this example ragged-right typesetting is used.

```
\usepackage{multicol,ragged2e}
\addtolength\columnsep{2pt}
\begin{multicols}{3}
\RaggedRight
\setcounter{unbalance}{1}
  Here is some text to be distributed over
  several columns. In this example ragged-right
  typesetting is used.
\end{multicols}
```

Column balancing is further controlled by the two counters columnbadness and finalcolumnbadness. Whenever LATEX is constructing boxes (such as a column) it will compute a badness value expressing the quality of the box—that is, the amount of excess white space. A zero value is optimal, and a value of 10000 is infinitely bad in LATEX's eyes.[2] While balancing, the algorithm compares the badness of possible solutions and, if any column except the last one has a badness higher than columnbadness, the solution is ignored. When the algorithm finally finds a solution, it looks at the badness in the last column. If it is larger than finalcolumnbadness, it will typeset this column with the excess space placed at the bottom, allowing it to come out short.

Collecting material

To be able to properly balance columns the multicols environment needs to collect enough material to fill the remaining part of the page. Only then does it cut the collected material into individual columns. It tries to do so by assuming that not more than the equivalent of one line of text per column vanishes into the margin due to breaking at vertical spaces. In some situations this assumption is incorrect and it becomes necessary to collect more or less material. In such a case

[1] Very bad for reading but too good to fix: this problem of a break-stack with "the" four times in a row will not be detected by TEX's paragraph algorithm—only a complete paragraph rewrite would resolve it.

[2] For an overfull box the badness value is set to 100000 by TEX, to mark this special case.

you can adjust the default setting for the counter collectmore. Changing this counter by one means collecting material for one more (or less) \baselineskip.

There are, in fact, reasons why you may want to reduce that collection. If your document contains many footnotes and a lot of surplus material is collected, there is a higher chance that the unused part will contain footnotes, which could come out on the wrong page. The smallest sensible value for the counter is the negative number of columns used. With this value multicols will collect exactly the right amount of material to fill all columns as long as no space gets lost at a column break. However, if spaces are discarded in this set up, they will show up as empty space in the last column.

Tracing the algorithm

You can trace the behavior of the multicol package by loading it with one of the following options. The default, errorshow, displays only real errors. With infoshow, multicol becomes more talkative and you will get basic processing information such as

```
Package multicol: Column spec: 185.0pt = indent + columns + sep =
(multicol)        0.0pt + 3 x 55.0pt + 2 x 10.0pt on input line 32.
```

which is the calculated column width.

With balancingshow, you get additional information on the various trials made by multicols when determining the optimal column height for balancing, including the resulting badness of the columns, reasons why a trial was rejected, and so on.

Using markshow will additionally show which marks for the running header or footer are generated on each page. Instead of using the options you can (temporarily) set the counter tracingmulticols to a positive value (higher values give more tracing information).

Manually breaking columns

Sometimes it is necessary to overrule the column-breaking algorithm. We have already seen how the unbalance counter is used to influence the balancing phase. But on some occasions one wishes to explicitly end a column after a certain line. In standard LaTeX this can be achieved with a \pagebreak command, but this approach does not work within a multicols environment because it will end the collection phase of multicols and thus end *all* columns on the page. As an alternative the command \columnbreak is provided. If used within a paragraph it marks the end of the current line as the desired breakpoint. If used between paragraphs it forces the next paragraph into the next column (or page) as shown in the following example. If \flushcolumns is in force, the material in the column is vertically stretched (if possible) to fill the full column height. If this effect is not desired one can prepend a \vfill command to fill the bottom of the column with white space.

Here is some text to be distributed over several columns.

With the help of the \columnbreak command this paragraph was forced into the second column.

3-5-18

```
\usepackage{multicol,ragged2e}
\begin{multicols}{2}  \RaggedRight
 Here is some text to be distributed over several
 columns. \par \vfill\columnbreak
 With the help of the \verb=\columnbreak= command
 this paragraph was forced into the second column.
\end{multicols}
```

Floats and footnotes in multicol

Floats (e.g., figures and tables) are only partially supported within `multicols`. You can use starred forms of the float environments, thereby requesting floats that span all columns. Column floats and `\marginpars`, however, are not supported.

Footnotes are typeset (full width) on the bottom of the page, and not under individual columns (a concession to the fact that varying column widths are supported on a single page).

Under certain circumstances a footnote reference and its text may fall on subsequent pages. If this is a possibility, `multicolst` produces a warning. In that case, you should check the page in question. If the footnote reference and footnote text really are on different pages, you will have to resolve the problem locally by issuing a `\pagebreak` command in a strategic place. The reason for this behavior is that `multicols` has to look ahead to assemble material and may not be able to use all material gathered later on. The amount of looking ahead is controlled by the `collectmore` counter.

3.5.4 changebar—Adding revision bars to documents

When a document is being developed it is sometimes necessary to (visually) indicate the changes in the text. A customary way of doing that is by adding bars in the margin, the known as “changebars”. Support for this functionality is offered by the changebar package, originally developed by Michael Fine and Neil Winton, and now supported by Johannes Braams. This package works with most PostScript drivers, but in particular dvips, which is the default driver when the package is loaded. Other drivers can be selected by using the package option mechanism. Supported options are `dvitoln03`, `dvitops`, `dvips`, `emtex`, `textures`, and `vtex`.

Supported printer drivers

```
\begin{changebar}[barwidth]     \cbstart[barwidth] ... \cbend
```

When you add text to your document and want to signal this fact, you should surround it with the `changebar` environment. Doing so ensures that LaTeX will warn you when you forget to mark the end of a change. This environment can be (properly) nested within other environments. However, if your changes start within one LaTeX environment and end inside another the environment form cannot be used as this would result in improperly nested environments. Therefore, the package also provides the commands `\cbstart` and `\cbend`. These should be used with care, because there is no check that they are properly balanced. Spaces after them might get ignored.

If you want to give a single bar a different width you may use the optional argument and specify the width as a normal LaTeX length.

\cbdelete [*barwidth*]

Text that has been removed can be indicated by inserting the `\cbdelete` command. Again, the width of the bar can be changed.

```
\usepackage{changebar}
\cbstart
This is the text in the first paragraph.
This is the text in the first paragraph.\cbend

This is the text in the second paragraph.
\cbdelete
This is the text in the second paragraph.

\setcounter{changebargrey}{35}
\begin{changebar}[4pt]
This is paragraph three. \par
This is paragraph four.
\end{changebar}
```

This is the text in the first paragraph. This is the text in the first paragraph.

This is the text in the second paragraph. This is the text in the second paragraph.

This is paragraph three.

This is paragraph four.

3-5-19

\nochangebars

When your document has reached the final stage you can remove the effect of using the changebar package by inserting the command `\nochangebars` in the preamble of the document.

Customizations

Changing the width If you want to change the width of *all* changebars you can do so by changing the value of `\changebarwidth` via the command `\setlength`. The same can be done for the deletion bars by changing the value of `\deletebarwidth` .

Positioning changebars By default, the changebars will show up in the “inner margin”, but this can be changed by using one of the following options: `outerbars`, `innerbars`, `leftbars`, or `rightbars`.

The distance between the text and the bars is controlled by `\changebarsep`. It can can be changed only in the preamble of the document.

Coloring changebars The color of the changebars can be changed by the user as well. By default, the option `grey` is selected so the changebars are grey (grey level 65%). The drivers `dvitoln03` and `emtex` are exceptions that will produce black changebars.

The “blackness” of the bars can be controlled with the help of the LaTeX counter `changebargrey`. A command like `\setcounter{changebargrey}{85}` changes that value. The value of the counter is a percentage, where 0 yields black bars, and 100 yields white bars.

The option `color` makes it possible to use colored changebars. It internally loads dvipsnames, so you can use a name when selecting a color.

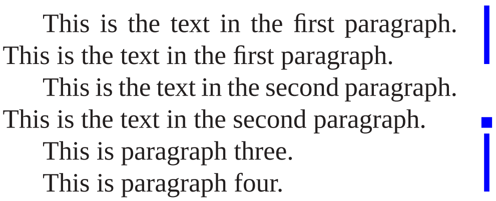
```
\cbcolor{name}
```

The color to use when printing changebars is selected with the command `\cbcolor`, which accepts the same arguments as the `\color` command from the color package [57, pp.317–326].

```
\usepackage[rightbars,color]{changebar}
\cbcolor{blue}
\setlength\changebarsep{10pt}
\cbstart
This is the text in the first paragraph.
This is the text in the first paragraph.\cbend

This is the text in the second paragraph.
\cbdelete
This is the text in the second paragraph.

\begin{changebar}
This is paragraph three. \par
This is paragraph four.
\end{changebar}
```



This is the text in the first paragraph. This is the text in the first paragraph.

This is the text in the second paragraph. This is the text in the second paragraph.

This is paragraph three.

This is paragraph four.

3-5-20

Tracing the algorithm

You can trace the behavior of the changebar package by loading it with one of the following options. The default, `traceoff`, displays the normal information LaTeX always shows. The option `traceon` informs you about the beginning and end points of changebars being defined. The *additional* option `tracestacks` adds information about the usage of the internal stacks.