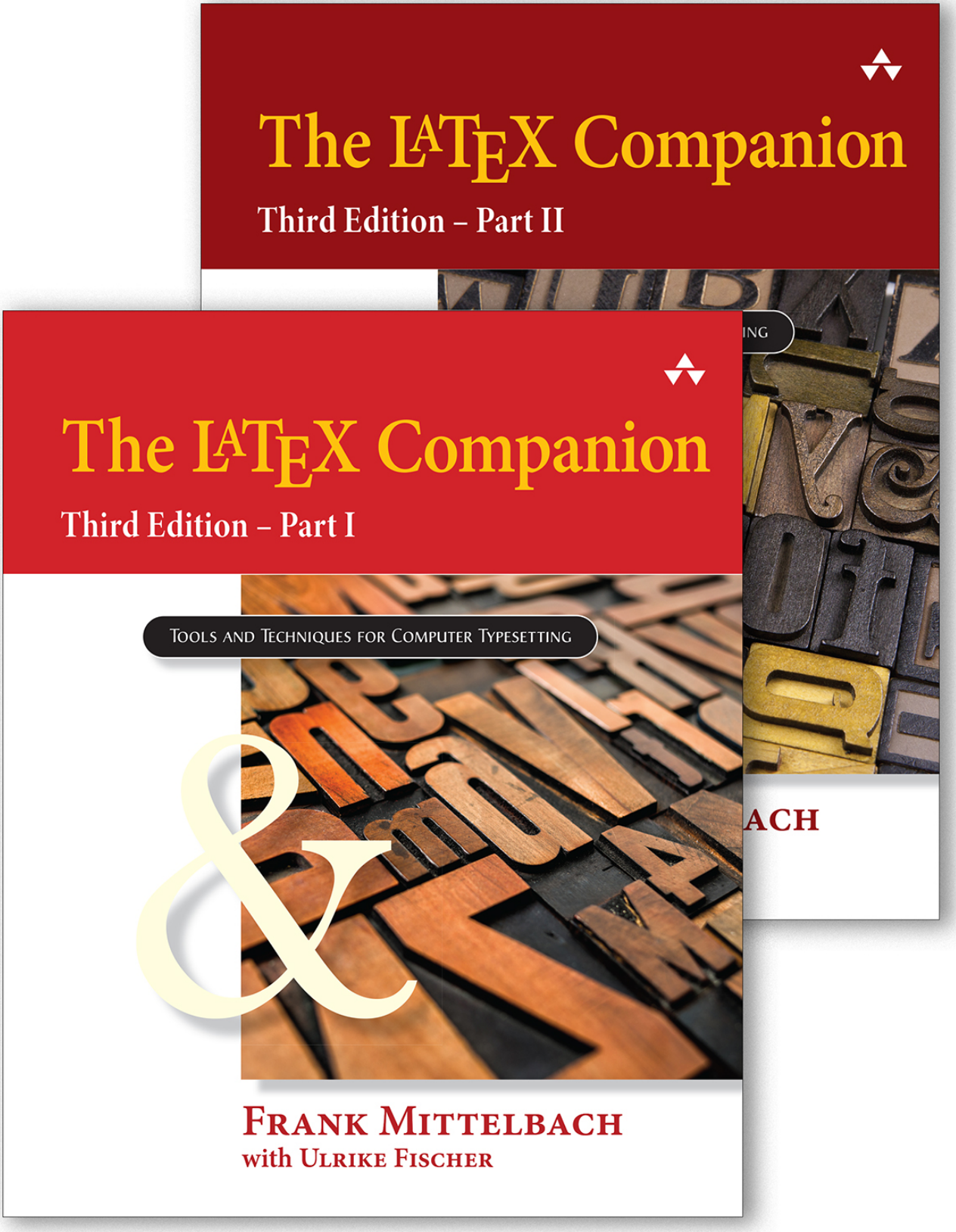

The LaTeX Companion
Third Edition – Part II
The LaTeX Companion
Third Edition – Part I
Tools and Techniques for Computer Typesetting
Frank Mittelbach
with Ulrike Fischer

The LaTeX Companion

Third Edition – Part I & Part II

This eBook is a compilation of Part I and Part II of *The LaTeX Companion*, Third Edition. To navigate to a specific page, click the links in the text or enter the part number, a hyphen, and the page number — e.g., `II-39` for page 39 in the second part.

Detailed information about the production of this eBook is given in the Production Notes on page →II 983.

Addison-Wesley Series on Tools and Techniques for Computer Typesetting

This series focuses on tools and techniques needed for computer typesetting and information processing with traditional and new media. Books in the series address the practical needs of both users and system developers. Initial titles comprise handy references for LaTeX users; forthcoming works will expand that core. Ultimately, the series will cover other typesetting and information processing systems, as well, especially insofar as those systems offer unique value to the scientific and technical community. The series goal is to enhance your ability to produce, maintain, manipulate, or reuse articles, papers, reports, proposals, books, and other documents with professional quality.

Ideas for this series should be directed to `frank.mittelbach@latex-project.org`. Send all other feedback to the publisher at `informit.com/about/contact_us` or via email to `community@informit.com`.

Series Titles

Guide to LaTeX, Fourth Edition by Helmut Kopka and Patrick W. Daly

The LaTeX Companion, Third Edition by Frank Mittelbach, with Ulrike Fischer and contributions by Javier Bezos, Johannes Braams, and Joseph Wright

The LaTeX Graphics Companion, Second Edition by Michel Goossens, Frank Mittelbach, Sebastian Rahtz, Denis Roegel, and Herbert Voß
Reprinted 2022 by Lehmanns Media, Berlin

The LaTeX Web Companion by Michel Goossens and Sebastian Rahtz

Also from Addison-Wesley and New Riders:

LaTeX: A Document Preparation System, Second Edition by Leslie Lamport

Computers & Typesetting, Volumes A–E by Donald E. Knuth

The Type Project Book: Typographic projects to sharpen your creative skills & diversify your portfolio by Nigel French and Hugh D'Andrade

The LaTeX Companion

Third Edition – Part I

Frank Mittelbach
LaTeX Project, Mainz, Germany

Ulrike Fischer
LaTeX Project, Bonn, Germany

With contributions by Joseph Wright

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Cover illustration by Lonny Garris/Shutterstock
Photos of Sebastian Rahtz courtesy of the TEX Users Group

Book design by Frank Mittelbach
Typeset with LATEX in Lucida Bright at 8.47pt/11.72pt

Library of Congress Control Number: 2022947208

Part I: Print ISBN-13: `978-0-13-465894-0`

Part II: Print ISBN-13: `978-0-201-36300-5`

Part I+II (bundled):
Print ISBN-13: `978-0-13-816648-9`

Part I+II (combined) digital:
ePub ISBN-13: `978-0-13-816652-6`
uPDF ISBN-13: `978-0-13-816657-1`

Release date of the digital edition: September 1, 2023

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at `https://www.pearson.com/report-bias.html`.

This page intentionally left blank

A TeX Haiku

```
\expandafter\def
\csname def\endcsname
{\message{farewell}}\bye
```

SPQR
at the poetry competition

TUG conference,
Vancouver, 1999

I dedicate this edition to all my friends in the TEX world and in particular to the memory of my good friend Sebastian P. Q. Rahtz (1955–2016), with whom I spent many happy hours discussing parenting, literature, LATEX and other important aspects of life [146].

This page intentionally left blank

Contents

Part I

Part II

List of Figures

Part I

Part II

List of Tables

Part I

Part II

Foreword

Before my retirement, I had the distinct privilege to work with leading authors in computing and related, technical fields. In many cases, my job as editor was simply to be an encouraging and sympathetic presence, as well as a welcome dining companion, while trying to make the publishing process as painless for them (and for in-house staff) as I could. As in childbirth, of course, eliminating all pain was virtually impossible; over unexpectedly long periods, authors yielded much too much time for family, pleasure, and sleep, all in the pursuit of a newborn book. I sometimes felt like an able midwife; other times, I could do nothing more than boil water and hope for the best. In the end, I was always proud of what these creative men and women could produce.

At no time during my lengthy tenure was my pride greater than it was for the authors who gave the world two, now three, editions of *The LaTeX Companion*. Building on the original inventions of Don Knuth and Leslie Lamport — speaking of my privilege to have worked with the best! — and led in each case by Frank Mittelbach, they have reached deeply into the work of selfless contributors, including themselves, to define the current state of LaTeX typesetting, and then to organize and document, in one authoritative and comprehensive publication, the tools now available for both beginning and advanced users.

My pride, I should say, has its origins in the book's publisher itself. Addison-Wesley (A-W), now an imprint of Pearson, had been founded by a printer, Melbourne Cummings, and Mel's values for production quality, particularly for textbooks with heavy mathematical content, were engrained in the company from the start (Thomas's *Calculus and Analytical Geometry* was his first book). Indeed, some notable authors with concern for the physical look of their books selected A-W precisely because of those values, even when they thought they might get a bigger

sales bang elsewhere (they ultimately were pleased to get both)! Don, by the way, having just developed TEX, was the author Mel most strongly insisted to me on meeting in person, wishing to speak, as it were, typesetter to typesetter.

I have to leave it for the Preface to describe the book's contents more specifically. I have been away from LATEX too long to be able to add much anyway. I have no idea, for example, whether newer versions of the system incorporate AI, so that a user might hear a HAL-like voice in the computer say something like, "Are you sure you want such narrow margins, Dave?" Nor do I know if the system now has 3D options, so that an important discovery literally jumps out from the page. Never mind. See the Preface.

What I can add from experience, and I am sure this much has not changed, is that LATEX authors and users are an intense and serious bunch when it comes to making their writing look good. I admire their attention to detail, to getting precisely the right format to present their ideas. I once was dining out with one such person, and watched as he studied the menu for quite some time. A very picky eater, I thought. But when he finally put the menu down, he tapped it with his pointed finger and told me, as the best appetizer for him, which letter didn't go well with the balance of the font. I, by contrast, soon became more concerned with a mushroom that didn't seem to go well with the rest of my meal.

From experience, too, I can tell you that there are LATEX users all over the world, and not just in those places you would expect to find them. The land of Gutenberg, sure, but how about a user in South America typesetting his book while bullets from a civil war literally flew by his university window (talk about intensity!)? I once also received user survey feedback from a urologist in Kenya. I frankly forget what his comment or question was — it was long ago — but I do remember being impressed how far LATEX use had spread, and into what surprising fields. Without doubt, an extensive literature search would turn up beautifully typeset works on the broadest range of topics, maybe even a book on digital rectal examinations.

Putting my own finger to the wind, as even former editors are wont to do, the need and demand for this revision are clear. Wherever you are, whatever your subject area, you will surely find in the pages (and pages) that follow the most helpful LATEX typesetting support a user could ever hope for. That certainly was my experience with the first two editions, and I now invite you to make it yours with the third.

Peter S. Gordon
Publishing Partner (Ret.)

To be continued in Part II . . .

Preface

With LaTeX being a voluntary effort,
it seems quite appropriate that
TLC also stands for “tender, loving care”
(Concise Oxford Dictionary)!
David Rhead, 1994

I have now been involved in computer based typesetting for nearly four decades, three of them as the technical lead for the development of LaTeX. During that long period there have been impressive technical advances in many different areas.

When I started there was no Internet to speak of — there were no browsers and there was no World Wide Web as we know it today. To book a hotel on my first trip to California to meet with Leslie Lamport, I had to resort to a travel agency that used fax machines to arrange the trip; on the flight I was served free alcoholic drinks (bad idea); and my computer at home was an Atari with two floppy drives (younger people probably only know these as the strange “save icon” in many software programs and perhaps have wondered what that represents) and an impressive external hard disc with 100mb of storage (that cost me a fortune at that time).

However, already back then LaTeX had existed for some time and worked fine, though a lot of today’s functionality was unavailable or, even if available, impossible to use, because computer processing speed was simply too slow.[1] As explained in more detail in the history section in Chapter 1, most of our enthusiastic ideas back then for a new and improved LaTeX were simply two decades too early, and while we had a fully working first version of the L3 programming layer in the early nineties our

[1] The first simple TeX documents I produced on a large university mainframe took about half a minute per page — you could literately watch the progress as `[`, *wait*, `1`, *wait*, `]`, *long wait*, … — and we still thought it was great and fast.

users would have died of caffeine consumption waiting for the results of processing their documents if we had dared to inflict it on them.

This all has changed since that time and today my smartphone is faster than the mainframe power available in the nineties. As a result, many new packages appeared over time and a lot of our dormant ideas and concepts envisioned in 1990 were finally integrated into LaTeX on the memorable day of February 2, 2020.

Since then, this programming environment has been used by the LaTeX Team to offer new functionality and also by many package authors developing new packages. All these developments — the recent as well as the older — are covered in this book.

The Companion *editions — setting the standard for a dozen years each*

When Michel, Alexander, and I wrote the first edition of *The LaTeX Companion* [56] in 1993, we intended to describe what is usefully available in the LaTeX world (though ultimately we ended up describing the then-new LaTeX 2ε standard and what was useful and available at CERN in those days). As an unintended side effect, this first edition *defined* for most readers what should be available in a then-modern LaTeX distribution. Fortunately, most of the choices we made at that time proved to be reasonable, and the majority (albeit not all) of the packages described in the first edition are still in common use today.

During the following decade the *Companion* (nicknamed the "doggie book" because of its cover) became a core resource for many LaTeX users, with several reprints and translations into German, Japanese, and Russian.

Our approach was to provide comprehensive coverage for typical LaTeX documents so that for most users the *Companion* would serve as the only reference needed to get "the job" done. More esoteric package features or features still under development were not described. Instead, pointers to the package documentation were given if we thought such a feature was worth mentioning. This approach worked well, so at the turn of the millennium one reviewer wrote, "while the book shows its age, it still remains a solid reference in most parts".

The second edition in the new millennium . . .

Nevertheless, much had changed and a lot of new and exciting functionality had been added to LaTeX during that decade and it became clear that a revised edition was necessary. This second edition [145], published in 2004, saw a major change in the authorship: I took over as principal author (so from then on I am to blame for all the faults in the *Companion* editions) and several members of the LaTeX Project Team joined in the book's preparation, enriching it with their knowledge and experience in individual subject areas.

We ended up rewriting 90% of the original content and adding about 600 additional pages describing impressive and useful new developments. As a result, the second edition was essentially a new book — a book that we hoped preserved the positive aspects of the first edition even as it greatly enhanced them, while at the same time avoiding the mistakes we made back then, both in content and presentation (though, of course, we made some new ones). From the reception in the user community, I think it is fair to say that we largely succeeded — in fact, that book served even longer as a useful resource.

… and nearly two decades later, the third

However, a decade or more is an awfully long time for a technical book, even given the longevity and stability of LaTeX and the *Companion*'s approach of describing a coherent and well-established set of packages. So in 2017 I started discussing with Kim Spenceley (my new editor at Addison-Wesley/Pearson after Peter Gordon's retirement) plans for a third edition of *The LaTeX Companion*. One question to solve up front was that of authorship. Initially, it looked as if I would have to do any necessary work all by myself this time, because none of the previous co-authors was available to help for one reason or another, making it a very daunting task indeed.

Fortunately, this impression was wrong! In the end I got great help from Ulrike Fischer, who wrote Chapters 15 and 16, the sections on hyperref and tikz, and helped with numerous tasks during the production of this edition.

Javier Bezos and Johannes Braams took on the task of revising Chapter 13 on localizing documents, and Joseph Wright helped with describing siunitx and the section on source control support. Thanks to all of them — without their help the book would have been be much more difficult to finish.

Big thanks to our volunteer copy editor Nelson!

Furthermore, Nelson Beebe kindly offered to read *all* chapters, checking them for accuracy as well as doing a first pass on copyediting. He provided numerous suggestions for improvements and I cannot thank him enough for undertaking that enormous task! The professional copy editor and the two proofreaders found additional boo-boos of mine, and I then found a few they missed while entering their corrections. I am sure our readers will find even more — it is a never-ending task, but we all did our best and, on the whole, I think we delivered a solid result.

The new edition

Initially, when I discussed plans for a third edition of *The LaTeX Companion*, I expected the need for a large number of updates to the existing material, but not many additions. Thus, my naive estimate was that the book would perhaps grow by 10–15%.

What's in it for you?

However, after researching in depth the new material that had been developed since 2004, it became crystal clear that to remain faithful to the core promise of the *Companion* — to be a solid reference for the majority of LaTeX users to get their work done — we had to include a much larger amount of new material:

- Descriptions of highly useful, large-scale packages that appeared in the meantime or were substantially updated in the last decade, e.g., biblatex, fontspec, hyperref, mathtools, siunitx, tcolorbox, unicode-math, and tikz, to name a few.
- A larger number of smaller packages that cover new ground and are useful for day-to-day work or for specialized (but not too esoteric) tasks.[1]
- Two new chapters on the exciting possibilities offered by using high-quality fonts for text *and* math — yes, LaTeX is no longer restricted to Computer Modern fonts or a few PostScript fonts that were set up for use with LaTeX in the nineties.

  You can now choose from a large number of high-quality, free fonts for both text and math; the only serious remaining problem is finding the ones you like. These chapters help with that, by showing samples of more than one hundred

[1] Give or take the odd exception, e.g., sillywalk, which I found just too lovely to bypass.

Big thanks to Adam, helping me with my two favorite "coffee table book" chapters

text fonts and more than forty alternative math font setups. I am very grateful for the help I received from Adam Twardoch (president of GUST, the Polish TEX users group, and the designer of the Lato fonts) on this, who spent many hours with me during two BachoTEX conferences, guiding me through the fonts available today and helping to select those of high quality for inclusion in the book.

- We also had to cover newer engine developments, e.g., the use of Unicode engines with LATEX, across all chapters of the book. There are often subtle differences that you need to be aware of if you use these engines.
- Finally, there have been very important changes to LATEX itself, which is undergoing a transformation that started in 2018, to keep it relevant in the years to come. Examples are the new hook management system for LATEX, the extended document command syntax, and the inclusion of the L3 programming layer into the LATEX format. All this is covered in the appropriate places — just take a peek at the term "L3 programming layer" in the index to see how much of the new material is already based on it.

  In that sense the third edition is like the first: both have been written just after LATEX itself had seen major changes and these exciting changes and additions are covered.

All this relevant information is now part of the new edition, but as a result we ended up with 1700 pages, not including the index — clearly too much to be printed as a single book that you can reasonably use as a day-to-day reference on your desk.

Two parts — one (virtual) book

For that reason the decision was made to split the book into two parts of roughly equal size and market them as a unit.[1] The chapter progression follows more or less the successful order of the earlier editions, starting with elements and concepts that you need quite often in nearly all documents (the first few chapters in Part I) followed by topics you also usually need in most documents but not necessarily all the time (remainder of Part I and most of Part II).

Of course, if you are a mathematician you might end up keeping Chapter 11 open all the time, but if your interest is typesetting novels or company reports, it might be the chapter least often touched.

Part II also contains three important appendices on core LATEX commands for defining your own little commands or applications, one on resolving errors (not that you would make any, would you?), and one on getting further help if this edition is not answering your questions — unlikely I'm sure, but then who knows?

As David Rhead observed in the quotation at the beginning of the preface, TLC is not only an acronym for *The LATEX Companion* but also stands for "tender, loving care" and this is most certainly an accurate description of the efforts and lifeblood poured into the works described on the pages of this book.

[1] For technical and accounting reasons that still means three separate ISBNs, one for each part and one for the bundle. From my perspective this is far from perfect, but it is how the world of publishing and logistics works. It means that while it is theoretically possible to buy only one part, there is little sense in that — unless you have used it so much that it is worn out and you want a fresh copy.

There are millions of LaTeX users out there (the online service Overleaf alone reported ten million accounts in 2022) and most of them use LaTeX because they love its typesetting capabilities and its superior quality. With LaTeX being easily extensible, users often adapt LaTeX to their needs in new fields and for new applications, and some of them go one step further, packaging their solution and making it available to others — usually supporting it long after they have a private need for it.

A standing salute to all these dedicated developers in the LaTeX world!

This is why we now have close to 5000 packages for use with LaTeX on the Comprehensive TeX Archive Network (CTAN) and why its catalogue lists nearly 3000 contributors. It is because of their dedication and "tender, loving care" that TeX and LaTeX stayed relevant for nearly four decades, offering unsurpassed quality and, likely, continuing to do so for several decades to come.

Looking back, it took roughly five years and several thousand hours to write this book, which sounds like an awfully long time and a huge effort — both of which are true — but the effort was rooted in the complexity and size of the task.

Research

The first phase of the production was reading through the documentation of nearly 5000 packages available in today's LaTeX distributions, classifying them according to functionality, usability, and correctness. This included testing all packages initially considered as candidates for inclusion, to see if their documentation actually matched reality (often it did not — mine included ☹) and to come up with relevant use cases and examples. Often, alternative solutions provided by different packages existed, in which case a more in-depth analysis was necessary to decide which packages to recommend. That phase took somewhat more than a year.

Describe

After this initial survey I started with documenting the selected packages or, in the case of packages already in the previous edition, revising and updating the existing material, describing new functionality, or rearranging the documentation to provide better access.

Frequently, while thinking up useful examples, I found some errors in a package or in its documentation or identified valuable but missing functionality, in which case a discussion with the package author started. As a side effect, this process more than once messed up the text that I had already written about the package, because afterwards I had to account for the new or changed functionality that I had requested. Thus, in several cases I rewrote whole sections or provided new, improved examples when further features became available. However, a real headache proved to be the larger, complex packages that sometimes come with several hundred pages of documentation. The task in such a case is to work through all this material and figure out what from it is needed by the majority of our readers, describe it adequately, and point out which areas I had left out or only skimmed over.

Of course, in many other cases the situation was reversed; i.e., the package functionality was good, but the documentation difficult to understand or incomplete, so here the task was to provide a different, possibly expanded, and hopefully better description. In either case, a strong focus was on providing useful, ready-to-apply examples, of which this edition has more than 1550. They have all been handcrafted to cover the typical use cases and support the accompanying documentation in the book.

This phase took close to three years, which means roughly writing two pages per day if working without break — going at it each and every day, including weekends (and for large periods it was like this).

Produce

The final phase, which started in spring 2022, was to pass all the work, chapter by chapter, to the professional copy editors engaged by Pearson, enter their corrections, and then do the layout of the chapters.[1] Once a chapter was in its final form I passed it back to a proofreader, who verified that the corrections had been correctly entered (not always), followed by a final pass by another proofreader who checked the final version once more. In parallel, Keith Harrison and I worked through all chapters to compile a useful concept index. The overall process took nine months, and I completely underestimated the amount of work necessary for this, even though I should have known better from previous books.

Many thanks to Kim and Julie for making the book a reality

My sincere thanks to Kim Spenceley, my editor at Pearson, and Julie Nahil, my senior content producer, who steered me patiently through the whole process, putting up with my idosyncrasies while keeping me on track, and at the same time making sure that my quest for quality was supported as much as possible.

Was it worth the effort?

Maybe you are asking yourself was that worth it, in the days of the Internet, where you can search for almost anything in a matter of seconds or watch a video that explains how to do something?

My personal answer to that question is a clear yes, because while there is a huge amount of information out there, it is of very varying quality, from extremely good to horrendously bad, misleading, or even plain wrong. This makes it very difficult for a user to sort the wheat from the chaff and, as a result, this overflow of information is not helpful unless you get good guidance.

And this guidance, we trust, is what the *Companion* is offering you, by providing you with a curated set of packages covering all areas of document production, showing you suitable solutions to various problems, and explaining the limitations when using one or the other approach.

We hope that this new edition will become a good companion for you for many years to come — just like the previous editions in the last decades. If it turns out that we achieved that, it will certainly be something to be proud of.

Frank Mittelbach

November 2022

[1] The nightmarish but also oddly satisfying task to lay out a book like this is described in the Production Notes at the very end of Part II.

CHAPTER 1

Introduction

LaTeX is not just a system for typesetting mathematics. Its applications span one-page memoranda, business and personal letters, newsletters, articles, and books covering the whole range of the sciences and humanities … right up to full-scale expository texts and reference works on all topics. Versions of LaTeX now exist for practically every type of computer and operating system. This book provides a wealth of information about its many present-day uses but first provides some background information.

The first section of this chapter looks back at the origins and subsequent development of LaTeX.[1] The second section gives an overview of the file types used by a typical current LaTeX system and the rôle played by each. Finally, the chapter offers some guidance on how to use the book.

1.1 A brief history (of nearly half a century)

In the Beginning …

In May 1977, Donald Knuth of Stanford University [95] started work on the text-processing system that is now known as "TeX and METAFONT" [84–88]. In the foreword of *The TeXbook* [84], Knuth writes: "TeX [is] a new typesetting system intended for the creation of beautiful books — and especially for books that contain a lot of mathematics. By preparing a manuscript in TeX format, you are telling a computer exactly how the manuscript is to be transformed into pages whose typographic quality is comparable to that of the world's finest printers."

[1] A more personal account can be found in *The LaTeX legacy: 2.09 and all that* [176].

In 1979, Gordon Bell wrote in a foreword to an earlier book, *TEX and METAFONT, New Directions in Typesetting* [82]: "Don Knuth's Tau Epsilon Chi (TEX) is potentially the most significant invention in typesetting in this century. It introduces a standard language in computer typography and in terms of importance could rank near the introduction of the Gutenberg press."

In the early 1990s, Donald Knuth produced an updated version and also officially announced that TEX would not undergo any further development [96, 97] in the interest of stability. Perhaps unsurprisingly, the 1990s saw a flowering of experimental projects that extended TEX in various directions; many of these are coming to fruition in the early 21st century, making it an exciting time to be involved in automated typography.

The development of TEX from its birth as one of Don's "personal productivity tools" (created simply to ensure the rapid completion and typographic quality of his then-current work on *The Art of Computer Programming*) [90] was largely influenced and nourished by the American Mathematical Society on behalf of U.S. research mathematicians.

... and Lamport saw that it was Good.

While Don was developing TEX, in the early 1980s, Leslie Lamport started work on the document preparation system now called LATEX, which used TEX's typesetting engine and macro system to implement a declarative document description language based on that of a system called Scribe by Brian Reid [168]. The appeal of such a system is that a few high-level LATEX declarations, or commands, allow the user to easily compose a large range of documents without having to worry much about their typographical appearance. In principle at least, the details of the layout can be left for the document designer to specify elsewhere.

The second edition of *LATEX: A Document Preparation System* [106] begins as follows: "LATEX is a system for typesetting documents. Its first widely available version, mysteriously numbered 2.09, appeared in 1985." This release of a stable and well-documented LATEX led directly to the rapid spread of TEX-based document processing beyond the community of North American mathematicians.

LATEX was the first widely used language for describing the logical structure of a large range of documents and hence introducing the philosophy of logical design, as used in Scribe. The central tenet of "logical design" is that the author should be concerned only with the logical content of his or her work and not its visual appearance. Back then, LATEX was described variously as "TEX for the masses" and "Scribe liberated from inflexible formatting control". Its use spread very rapidly during the next decade. By 1994 Leslie could write, "LATEX is now extremely popular in the scientific and academic communities, and it is used extensively in industry". But that level of ubiquity looks quite small when compared with the present day when it has become, for many professionals on every continent, a workhorse whose presence is as unremarkable and essential as the workstation on which it is used.

Going global

The worldwide availability of LATEX quickly increased international interest in TEX and in its use for typesetting a range of languages. LATEX 2.09 was (deliberately) not globalized, but it was globalizable; moreover, it came with documentation worth translating because of its clear structure and straightforward style. Two pivotal conferences (Exeter UK, 1988, and Karlsruhe Germany, 1989) established clearly the widespread adoption of LATEX in Europe and led directly to International LATEX [180]

and to work led by Johannes Braams [23] on more general support for using a wide variety of languages and switching between them (see Chapter 13).

Note that in the context of typography, the word *language* does not refer exclusively to the variety of natural languages and dialects across the universe; it also has a wider meaning. For typography, "language" covers a lot more than just the choice of "characters that make up words", as many important distinctions derive from other cultural differences that affect traditions of written communication. Thus, important typographic differences are not necessarily in line with national groupings but rather arise from different types of documents and distinct publishing communities.

The Next Generation

Another important contribution to the reach of LaTeX was the pioneering work of Frank Mittelbach and Rainer Schöpf on a complete replacement for LaTeX's interface to font resources, the New Font Selection Scheme (NFSS) (see Chapter 9). They were also heavily involved in the production of the $\mathcal{A}_{\mathcal{M}}\mathcal{S}$-LaTeX system that added advanced mathematical typesetting capabilities to LaTeX (see Chapter 11).

As a reward[1] for all their efforts, which included a steady stream of bug reports (and fixes) for Leslie, by 1989 Frank and Rainer "were allowed" to take over the maintenance and further development of LaTeX. One of their first acts was to consolidate International LaTeX as part of the kernel[2] of the system, "according to the standard developed in Europe". Very soon version 2.09 was formally frozen, and although the change-log entries continued for a few months into 1992, plans for its demise as a supported system were already far advanced as something new was badly needed.

Too much of a Good Thing™

The worldwide success of LaTeX had by the early 1990s led in a sense to too much development activity: under the hood of Leslie's "family sedan" many TeXnicians had been laboring to add such goodies as super-charged, turbo-injection, multivalved engines and much "look-no-thought" automation. Thus, the announcement in 1994 of the new standard LaTeX, christened LaTeX 2_ε, explains its existence in the following way:

> Over the years many extensions have been developed for LaTeX. This is, of course, a sure sign of its continuing popularity but it has had one unfortunate result: incompatible LaTeX formats came into use at different sites. Thus, to process documents from various places, a site maintainer was forced to keep LaTeX (with and without NFSS), SLiTeX, $\mathcal{A}_{\mathcal{M}}\mathcal{S}$-LaTeX, and so on. In addition, when looking at a source file it was not always clear for which format the document was written.
>
> To put an end to this unsatisfactory situation a new release of LaTeX was produced. It brings all such extensions back under a single format and thus prevents the proliferation of mutually incompatible dialects of LaTeX 2.09.

Standard LaTeX (LaTeX 2_ε)

The development of this "New Standard LaTeX" and its maintenance system was started in 1993 by the LaTeX Project Team [148], which soon comprised the author of this book, Rainer Schöpf, Chris Rowley, Johannes Braams, Michael Downes (1958–2003), David Carlisle, Alan Jeffrey, and Denys Duchier, with some encouragement and gentle bullying from Leslie. Although the major changes to the basic LaTeX system (the kernel) and the standard document classes (styles in 2.09) were completed by

[1] Pronounced "punishment".

[2] *Kernel* here means the core, or center, of the system.

1994, substantial extra support for colored typography, generic graphics, and fine positioning control were added later, largely by David Carlisle. Access to fonts for the new system incorporated work by Mark Purtill on extensions of NFSS to better support variable font encodings and scalable fonts [30–32].

1994 — The first edition of the LaTeX Companion

At this point in the story the first edition of the *LaTeX Companion* was written, which helped a lot in making many important packages known to a wide audience and as a side effect helped shape a standard corpus of LaTeX packages expected to be available on any installation across the world.

Towards the 21st century

Although the original goal for this LaTeX 2ε was consolidation of the wide range of incompatible models carrying the LaTeX marquee, what emerged was a substantially more powerful system with both a robust mechanism (via LaTeX packages) for extension and, importantly, a solid technical support and maintenance system. This provides robustness via standardization and maintainability of both the code base and the support systems. The core of this system remains the current standard LaTeX system that is described in this book. It has fulfilled most of the goals for "a new LaTeX for the 21st Century", as they were envisaged back in 1989 [151, 153].

The specific claims of the current system are "… better support for fonts, graphics and color; actively maintained by the LaTeX Project Team". The details of how these goals were achieved, and the resulting subsystems that enabled the claims to be substantially attained, form a revealing study in distributed software support: the core work was done in at least five countries and, as is illustrated by the bugs database [108], the total number of active contributors to the technical support effort remains high.

The package system

Although the LaTeX kernel suffered a little from feature creep in the late 1990s, the package system together with the clear development guidelines and the legal framework of the LaTeX Project Public License (LPPL) [111, 132] have enabled LaTeX to remain almost completely stable while supporting a wide range of extensions. These have largely been provided by a similarly wide range of people who have, as the project team are happy to acknowledge and the online catalogue [197] bears witness, enhanced the available functionality in a vast panoply of areas.

Development work

All major developments of the base system have been listed in the regular issues of *LaTeX News* [107]. At the turn of the century, development work by the LaTeX Project Team focused on the following areas: supporting multi-language documents [130]; a "Designer Interface for LaTeX" [141]; major enhancements to the output routine [131]; improved handling of inter-paragraph formatting; and the complex front-matter requirements of journal articles. Back then prototype code had been made available (see [140]), but the work has otherwise been kept separate from LaTeX — partly because it was executing simply too slowly on the available hardware.

No new features at the kernel level …

One thing the project team steadfastly refused to do at that time was to unnecessarily "enhance" the kernel by providing additional features as part of it, thereby avoiding the trap into which LaTeX 2.09 fell in the early 1990s: the disintegration into incompatible dialects where documents written at one site could not be successfully processed at another site. In this discussion it should not be forgotten that LaTeX serves not only to produce high-quality documents but also to enable collaboration and exchange by providing a lingua franca for various research communities.

With LaTeX 2ε, documents written in 1996[1] can still be run with today's LaTeX. In the opposite direction, new documents run on older kernel releases if the additional packages used are brought up-to-date — a task that, in contrast to updating the LaTeX kernel software, is easily manageable even for users working in a multiuser environment (e.g., in a university or company setting).

… but no standstill

But a stable kernel is not identical to a standstill in software development; of equally crucial importance to the continuing relevance and popularity of LaTeX is the diverse collection of contributed packages building on this stable base. The success of the package system for nonkernel extensions is demonstrated by the enthusiasm of these contributors — many thanks to all of them! As can be easily appreciated by visiting the highly accessible and stable Comprehensive TeX Archive Network (see Appendix C) or by reading this book (where more than 250 of these "Good Guys"[2] are listed on page →II 967), this has supported the existence of an enormous treasure trove of LaTeX packages and related software.

The back office

The provision of services, tools, and systems-level support for such a highly distributed maintenance and development system was itself a major intellectual challenge, because many standard working methods and software tools for these tasks assume that your colleagues are in the next room, not the next continent (and in the early days of the development, e-mail and FTP were the only reliable means of communication). The technical inventiveness and the personalities of everyone involved were both essential to creating this example of the friendly face of open software maintenance, but Alan Jeffrey and Rainer Schöpf deserve special mention for "fixing everything".

A vital part of this system that is barely visible to most people is the regression testing system with its vast suite of test files [129]. It was initially devised and set up by Frank and Rainer with Daniel Flipo; it has proved its worth countless times in the never-ending battle with the bugs. Over the years it has seen many refinements, cumulating in a complete rewrite as part of l3build [147], which we describe in Section 17.3 on page →II 606.

2004 — The second edition of the LaTeX Companion

In 2004, i.e., roughly a decade after its first edition, the second edition of the *LaTeX Companion* was published. Due to the popularity of LaTeX 2ε and its extended features for developers, new important packages had emerged, and LaTeX had reached out into new domains. While the advice given in the first edition remained largely valid (last but not least because of the long-term backward compatibility paradigm of LaTeX), we ended up rewriting 90% of the original content and added about 600 pages to account for new developments. As before, the second edition helped a lot in standardizing the use, and this way the interoperability, of LaTeX across the world.

Research

Some members of the LaTeX Project Team have built on the team's experience to extend their individual research work in document science beyond the current LaTeX structures and paradigms. Some examples of their work up to now can be found

[1] The time between 1994 and 1996 was a consolidation time for LaTeX 2ε, with major fixes and enhancements being made until the system was thoroughly stable. In fact, with some minor alterations in pagination or font usage, it is usually possible to reprocess even documents from the eighties (i.e., written for LaTeX 2.09) or make them reusable with little effort.

[2] Unfortunately, this is nearly the literal truth: you need a keen eye to spot the few ladies listed.

in the following references: [33, 35–37, 133–135, 138, 149, 175, 177]. An important spin-off from the research work was the provision of some interfaces and extensions that are immediately usable with standard LaTeX.

. . . and into the future

The decision to keep the core of the standard LaTeX system stable and essentially unchanging had two major advantages over any other approach to support fully automated document processing. First, the system already efficiently provided high-quality formatting of a large range of elements in very complex documents of arbitrary size. Second, it was robust in both use and maintenance and hence offered the potential to remain in widespread use for at least a further 15 years.[1] In the second edition of this book we wrote on this topic:

> As more such functionality is added, it will become necessary to assess the likelihood that merely extending LaTeX in this way will provide a more powerful, yet still robust and maintainable, system. This is not the place to speculate further about the future of LaTeX but we can be sure that it will continue to develop and to expand its areas of influence whether in traditional publishing or in electronic systems for education and commerce.

Reassessment time

This reassessment became necessary in the second decade of the new century, when it became obvious that this position was gradually getting unsustainable, because more and more areas in which people were looking for solutions could not be adequately addressed with a model of a fixed kernel and all developments outsourced to the package level. Examples are the move to Unicode in basically all operating systems and the growing pressure to produce "accessible" documents that conform to standards such as PDF/UA (Portable Document Format/Universal Accessibility).

An important policy change

Thus, in 2015, the LaTeX Project Team changed its policy and restarted kernel development. To retain the best of both worlds this was accompanied by developing a rollback/roll-forward functionality for the kernel and packages (that care to implement it). This allows a current LaTeX format to roll back to an earlier point in time in order to process old documents that rely on interfaces that have been changed since then or to process documents that explicitly worked around bugs (and so expect them to be there) that have been fixed in the meantime.

The first action of the team was to retire the `fixltx2e` package and instead include the accumulated fixes it contained directly in the format and to officially support LaTeX when using the Unicode engines XeTeX and LuaTeX. A big step forward happened in 2018 when LaTeX switched its default input encoding to UTF-8. This change proved that the policy change was the right thing to do and that the preparatory work (e.g., providing rollback) allows executing even major changes without disruption in its user base in order to keep LaTeX relevant and useful. A good indicator for the renewed and increased activity are the regular LaTeX newsletters [107] accompanying each release, which grew bulkier and again appeared semi-annually.

[1] One of the authors of the second edition had publicly staked a modest amount of beer on TeX remaining in general use (at least by mathematicians) until at least 2010. He should have made a larger bet, given that this is now 2022 and LaTeX is healthy and in fact growing its user base due to its many unsurpassed qualities.

And where is the mythical LaTeX3?

The event of providing the mythical LaTeX3 had long become a standing joke as "two years from 'now' — with 'now' a moving target". The reason was that the concepts and ideas for LaTeX3 have been simply a decade or more too early, and while the team implemented a fully working version already in 1990, it was simply too slow to be usable with the then available computing power. Thus, we gave up pursuing it and instead concentrated on offering LaTeX 2ε, which then went public in 1994.

But ideas and concepts were never forgotten by the team, and especially its newer members (who joined in this century) pushed them back to the forefront and improved them dramatically. As a result, the code was eventually publicly made available as the expl3 package. It was then picked up by a number of enthusiastic package developers and used as the basis for their new packages. For example, if you use acro, breqn, fontspec, siunitx, unicode-math, or xparse, to name a few, you use "LaTeX3" under the hood; a recent count shows more than 200 such packages or classes as part of TeX Live.

. . . well it got merged into the kernel in 2020

So in 2019 the LaTeX Project Team made two wide-ranging decisions: there will not be a separate LaTeX3 that is being developed alongside LaTeX 2ε (as was originally planned). Instead, we will modernize the current LaTeX gradually from the inside, using the new rollback mechanism and "development" formats as a safety net to ensure that there is no disruption of service for our user base. As a first step on this journey, the L3 programming layer and the LaTeX3 document-level command declarations (formerly known as expl3 and xparse) were made an integral part of LaTeX on February 2, 2020. Thus, more or less exactly 30 years after its conception, LaTeX3 became a reality for every LaTeX user — even though few will have immediately noticed.

The foundation layer for modernization

The importance of this step is that it allows the team to modernize other parts of the kernel and develop new functionality entirely based on the L3 programming layer, which offers many features not available with legacy LaTeX programming constructs. For example, the new Hook Management System for LaTeX, which is a cornerstone for modernizing and transforming the existing LaTeX, is entirely written using the new L3 programming layer, and other parts will follow suit.

Today's challenge: structured and accessible output is needed

As already mentioned, there is a steadily increasing interest in the production of "tagged" PDF documents that are "accessible", in the sense that they contain information to assist screen reading software, etc., and, more formally, that they adhere to the PDF/UA (Portable Document Format/Universal Accessibility) standard [190], explained further in [47]. In many disciplines this is starting to become a requirement when applying for grants or when publishing results.

At the moment, all methods of producing such "accessible PDFs", including the use of LaTeX, require extensive manual labor in preparing the source or in post-processing the PDF (maybe even at both stages); and these labors often have to be repeated after making even minimal changes to the (LaTeX or other) source. This is a huge pity, because LaTeX should in theory be well positioned to do this work automatically, given that its source is already well-structured.

The production of tagged (i.e., structured) PDF documents is not only important in order to comply to accessibility standards. It also opens possibilities to reuse data from such PDFs, because it allows other applications to correctly identify the structure inside the output document and this way extract or manipulate parts of the content — workflows that become increasingly important in the digital world.

The LaTeX Project Team has for some years been well aware that these new usages are not adequately supported by the current system architecture of LaTeX 2ε and that major work in this area is therefore urgently needed to ensure that LaTeX remains an important and relevant document source format. However, the amount of work required to make such major changes to the LaTeX system architecture is enormous and definitely way beyond the limited resources of a small team of volunteers working in their spare time (or maybe just about possible, but only given a very long — and most likely too long — period of time).

A multi-year project to shape the future of LaTeX

At the TeX Users Group conference 2019 in Palo Alto the team's previously pessimistic outlook on this subject became cautiously optimistic, because of discussions with senior executives from Adobe about the possibility of producing structured PDF from LaTeX source without the need for the usual requirement of considerable manual post-processing. As a result of these discussions, towards the end of 2019 the team produced an extended feasibility study for the project, aimed primarily at Adobe engineers and decision-makers. This study [144] describes in some detail the various tasks that constitute the project and their interdependencies. It also contains a project plan covering how, and in what order, these tasks should be tackled both to achieve the final goal and, at the same time, to provide intermediate concrete results that are relevant to user communities (both LaTeX and PDF); these intermediate results will help in obtaining feedback that is essential to the successful completion of later tasks.

This multi-year project found the approval of Adobe, which then committed to financially and otherwise supporting this endeavor [150]. Unfortunately — thanks to the COVID-19 pandemic — the start got delayed, but since the end of 2020, this exciting project is now well under way. First results from this project that are already in existence (such as the new hook management system and the alignment of the hyperref package with the LaTeX kernel) are already described in this book. Other parts are obviously still vaporware at this point. Fortunately, none is expected to render any documentation or suggestion made in this book obsolete — after all, the project goal is to enable tagging of existing documents, simply by reprocessing with minor configuration changes as outlined in the "Spoiler alert" Section 2.1.1 on page 23.

1.2 Today's systems

When we wrote the second edition of *The LaTeX Companion* (i.e., 2003–2004), standard LaTeX was (officially) supported only on 8-bit engines, e.g., pdfTeX. Around the same time, the first version of the Unicode engine XeTeX and (somewhat later, in 2007) the first beta version of LuaTeX appeared, and there were soon unofficial support files that helped people running LaTeX on these Unicode engines as well.

When LuaTeX reached version 1.0, the LaTeX Project Team used the opportunity and officially took on LaTeX support for all three engines that included, for example, running the release regression test suite with its roughly 1000 tests against all three engines. Besides these three engines (which are covered in this book), there are further ones, such as pTeX and upTeX for Japanese, where the LaTeX adjustments for the engine are maintained by the respective user groups.

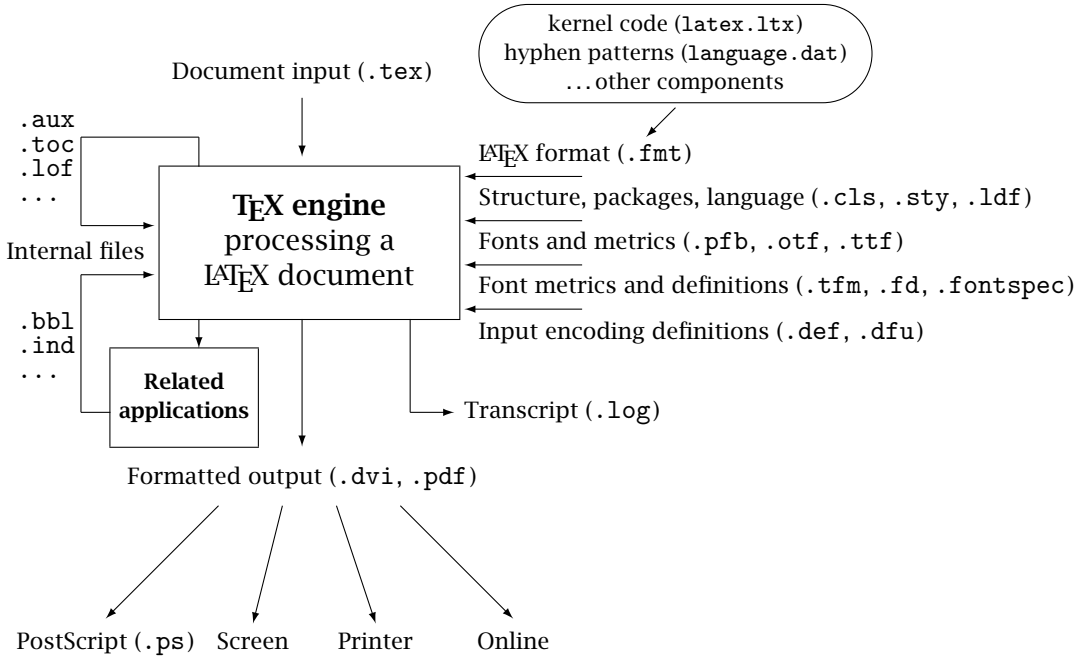


Figure 1.1: Data flow in the LaTeX system

What is described in this book should work with all of these engine — in cases where there are differences between 8-bit and Unicode engines, then they are explicitly described (see page 18 for a description how).

Engine specifics not covered in this book

However, each of the engines also has one or the other specialty compared to the original TeX program, which is available only with that particular engine; e.g., LuaTeX supports code written in Lua, or upTeX offers special commands for Japanese typography, etc. Standard LaTeX either abstracts such features (when support is available in all engines and only the methods differ) or does not make use of the features — and for that reason such engine-specific commands are not discussed in the *LaTeX Companion*. If you are interested in that level of coding, please refer to the engine documentation, e.g., for pdfTeX [65], for XeTeX [173], and for LuaTeX [122].

Files used in the LaTeX universe

In the remainder of the current section we present an overview of the vast array of files used by a typical LaTeX system with its many components. This overview also involves some descriptions of how the various program components interact. Most users will never need to know the exact details of this software environment that supports their work, but this section will be a useful general reference and an aid to understanding some of the more technical parts of this book.

Although modern LaTeX systems are most often embedded in project-oriented, menu-driven interfaces, behind the scenes little has changed from the file-based description given here. Figure 1.1 shows schematically the flow of information.

The following description assumes familiarity with a standard computer file system in which a "file extension" is used to denote the "type of a file". In processing a document, the LaTeX program reads and writes several files, some of which are further processed by other applications. The most important ones are listed in Table 1.1 on the next page. The book covers a total of 64 file types, but those not described in the current section are rather specialized and used only by individual packages.

Document input

The most obviously important files in any LaTeX-based documentation project are the *input source files*. Typically, there will be a main file that uses other subsidiary files (see Section 2.1). These files most often have the extension `.tex` (code documentation for LaTeX typically carries the extension `.dtx`; see Chapter 17). They are commonly known as "plain text files", because they can be prepared with a basic text editor. Often, external graphical images are included in the typeset document utilizing the graphics interface described in Section 8.1.

Structure and style

LaTeX also needs several files containing structure and layout definitions: *class* files with the extension `.cls`; *option* files with the extension `.clo`; *package* files with the extension `.sty` (see Appendix A). Many of these are provided by the basic system setup, but others may be supplied by other developers. LaTeX is distributed with five standard document classes: article, report, book, slides, and letter. These document classes can be customized by the contents of other files specified either by class options or by loading additional packages as described in Section 2.1. In addition, many LaTeX documents automatically input *language definition files* of the babel system with the extension `.ldf` (see Chapter 13) and *encoding definition files* of the inputenc/fontenc packages with the extension `.def` (see Chapter 9).

Font resources

The information that LaTeX needs about the glyphs to be typeset is found in *TeX font metric* files (extension `.tfm`). This does not include information about the shapes of glyphs, only about their dimensions. Information about which font files are needed by LaTeX is stored in *font definition* files (extension `.fd`), or in case of Unicode engines sometimes in `.fontspec` files. Both types are loaded automatically when necessary. See Chapter 9 for further information about font resources.

The LaTeX format

A few other files need to be available to TeX, but you are even less likely to come across them directly. An example includes the LaTeX format file `pdflatex.fmt` that contains the core LaTeX instructions, precompiled for processing by the pdfTeX formatter. There are some situations in which this format needs to be recompiled — for example, when changing the set of hyphenation rules available to LaTeX (configured in `language.dat`; see Section 13.6.2) and, of course, when a new LaTeX kernel is made available. The details regarding how such formats are generated differ from one TeX implementation to the next, so they are not described in this book, but usually this all happens behind the scenes with the tools of the distribution you use.

The output from LaTeX itself is a collection of *internal* files (see below), plus one very important file that contains all the information produced by TeX about the typeset form of the document.

Formatted output

TeX's own particular representation of the formatted document is that of a *device-independent* file (extension `.dvi`). TeX positions glyphs and rules with a precision far better than 0.01 μm (1/4,000,000 inch). Therefore, the output generated by TeX can be effectively considered to be independent of the abilities of any physical rendering device — hence the name. These days all major TeX engines can alternatively produce

	File Type	*Common File Extension(s)*
Document Input	text	`.tex .ltx`
	bibliography	`.bbl`
	index / glossary	`.ind` / `.gnd`
Graphics	internal	`.tex`
	external	`.ps .eps .tif .png .jpeg .jpg .gif .pdf`
Other Input	layout and structure	`.clo .cls .sty`
	encoding definitions	`.def .dfu`
	language definitions	`.ldf .ini`
	font access definitions	`.fd .fontspec`
	configuration data	`.cfg`
Internal Communication (Input and Output)	auxiliary	`.aux`
	table of contents / partial	`.toc` / `.ptc`
	list of figures / tables	`.lof` / `.lot`
Low-Level TEX Input	format	`.fmt`
	font image files	`.pfb .otf .ttf .pk`
	font metrics	`.tfm`
Output	formatted result	`.dvi .pdf`
	raw index / raw glossary	`.idx` / `.glo`
	transcript	`.log`
Bibliography (BIBTEX)	input / output	`.aux` / `.bbl`
	database / style / transcript	`.bib` / `.bst` / `.blg`
*(*biblatex*)*	style / citation / model / config	`.bbx` / `.cbx` / `.dbx` / `.bcf`
Index	input / output	`.idx` / `.ind`
	style / transcript	`.ist` / `.ilg`
Documentation & Testing	documentation / unpacking	`.dtx .fdd` / `.ins`
	test input / test output	`.lvt` / `.tlg`
Archive	dependencies / file usage	`.dep` / `.fls`

Table 1.1: Major file types used by LATEX

PDF output (extension `.pdf`), and over time this has become the standard output format largely replacing `.dvi`.[1] The `.dvi` file format specifies only the names/locations of fonts and their glyphs — it does not contain any rendering information for those glyphs. The `.pdf` file format can and usually does contain such rendering information.

Cross-references

Some of the *internal* files contain code needed to pass information from one LATEX run to the next, such as for cross-references (the *auxiliary* file, extension `.aux`; see Section 2.3) and for typesetting particular elements of the document such as the table of contents (extension `.toc`) and the lists of figures (extension `.lof`) and

[1] There are established workflows based on `.dvi` usually post-processed further to PostScript and from there often to PDF. For that reason the original format will remain a viable option.

of tables (extension `.lot`). Others are specific to particular packages (such as acro, Section 3.3.2, or enotez, Section 3.5.10) or to other parts of the system (see below).

Errors, warnings, and information

Finally, TEX generates a transcript file of its activities with the extension `.log`. This file contains a lot of information, such as the names of the files read, the page numbers (in brackets) of the pages processed, warning and error messages, and other pertinent data that is especially useful when debugging errors (see Appendix B). When you use an editor with integrated TEX support, this `.log` file is sometimes hidden from you and its data only selectively presented. If that is the case, it might be worth looking for it on the file system level, because it is likely to contain important information in case of problems that puzzle you.

Indexing

A file with the extension `.idx` contains individual unsorted items to be indexed. These items need to be sorted, collated, and unified by a program like *MakeIndex*, upmendex, or xindy (see Chapter 14). The sorted version is typically placed into a file (extension `.ind`) that is itself input to LATEX. For *MakeIndex* or upmendex, the *index style information* file has an extension of `.ist`, and the transcript file has an extension of `.ilg`; in contrast, xindy appears not to use any predefined file types.

Citations and bibliography

Information about bibliographic citations (see Chapter 16) in a document is normally output by LATEX to the *auxiliary* file or to the biber control file (extension `.bcf`). This information is used first to extract the necessary information from a bibliographic database and then to sort it; the sorted version is put into a *bibliography* file (extension `.bbl`) that is itself input to LATEX. If the system uses BIBTEX or biber (see Chapter 15) for this task, then the *bibliographic database* files will have an extension of `.bib`, and the transcript file will have the extension `.blg`. With BIBTEX, additional information about the process will be in a *bibliography style* file (extension `.bst`); biber does not use styles — this is handled by biblatex in that case.

Using `\specials` in the `.dvi` workflow

Because of the limitations of TEX, especially its failure to natively handle graphics or color, it is often necessary to complete the formatting of some elements of the typeset document after TEX has positioned everything and written this information to the `.dvi` file in some post-processing step. This is normally done by attaching extra information and handling instructions at the correct "geometrical position in the typeset document", using TEX's `\special` primitive that simply puts this information at the correct place in the `.dvi` file (see Chapter 8). This information may be simply the name of a graphics file to be input; or it may be instructions in a graphics language. This is then post-processed when the `.dvi` file is converted by a separate program, such as dvips, for printing or displaying. If TEX is directly generating PDF, there is conceptually not much difference, except that the post-processing happens directly in the extended TEX engine (e.g., pdftex, XETEX, or LuaTEX) at the point where TEX has finished a page and passes the result to a component that translates it to a PDF page. This component then plays the rôle that external programs play in the `.dvi` workflow: it also uses either `\specials` to communicate or additional primitives of the particular engine that do a similar job.

Using `\specials` in the `.pdf` workflow

In either case, LATEX abstracts from the underlying workflow peculiarities so that you can always just specify `\color` or `\includegraphics`. LATEX translates that into the right `\special` commands based on your workflow and the chosen TEX engine.

Seeing is believing

Once the document has been successfully processed by TEX (and possibly transformed into PostScript or PDF), you probably want to take a look at the formatted text.

This is commonly done on screen, but detailed inspection of printed output should always be performed via printing on paper at the highest available resolution. The applications available for viewing documents on screen vary quite a lot depending on your chosen workflow and your operating system. If you generate PDF, then various free and commercial tools exist that differ mainly in their features to post-process the document, but not in the actual representation, because the PDF normally includes all resources used by the document. If, on the other hand, you want to view a `.dvi` file, you need a viewer that can find and display the fonts or graphics referenced in the `.dvi`, because they are not part of the file itself. Occasionally you therefore find that some applications produce far superior screen output than others; this is due to limitations of the different technologies and the availability of suitable font resources.

1.3 Working with this book

This final section of Chapter 1 gives an overview of the structure of this edition, the typographic conventions used, and ways to use the examples given. Because of its size, this edition is typeset as two separate physical volumes (Part I and Part II), which has some implications on the presentation.

Chapters are numbered consecutively across both volumes, but we restart the page numbers in Part II to keep the numbers readable. As a consequence, cross-references to pages come in two forms: if they are to a page in the same volume, they read "see page 253", but if they refer to a page in the other volume, they look like "see page →II 127" or similar.

The main index, which contains entries for the whole edition, is replicated at the end of each physical volume to improve its usability and make it easier to work with. To identify the volume each page number in an entry refers to, the start of each volume sequence is identified by →I and →II, respectively.

1.3.1 What's where

Following is a summary of the subject areas covered by each chapter and appendix. In principle, all chapters can be read independently because, when necessary, pointers are given to where necessary supplementary information can be found in other parts of the edition.

Part I —

Chapter 1 gives a short introduction to the LaTeX system and this book.

Chapter 2 discusses document structure markup, including sectioning commands and cross-references as well as document source management.

Chapter 3 describes LaTeX's basic typesetting commands for the paragraph level. It also contains a section on packages offering document development support.

Chapter 4 looks at the typesetting of larger structures, such as lists and code displays, and shows how to work with multiple columns.

Chapter 5 explains how to influence the visual layout of the pages in various ways.

Chapter 6 shows how to lay out material tables, on single and multiple pages.

Chapter 7 surveys floating material and caption formatting.

Chapter 8 covers image loading and manipulation and the generation of portable graphics. It also offers an extensive overview on the tcolorbox package and an introduction to the world of tikz.

Chapter 9 discusses in detail LaTeX's Font Selection Scheme and shows how to access new fonts in 8-bit and Unicode TeX engines.

Part II —

Chapter 10 gives a comprehensive list with examples of high-quality text and symbol fonts available out of the box to LaTeX users today.

Chapter 11 reviews mathematical typesetting, particularly the packages supported by the American Mathematical Society.

Chapter 12 describes aspects of font usage in math formulas and offers a comparison between available font setups with 8-bit and Unicode TeX engines.

Chapter 13 discusses the support for using LaTeX with multiple languages, particularly the babel system.

Chapter 14 discusses the preparation and typesetting of an index with a focus on the programs *MakeIndex* and upmendex.

Chapter 15 explains how to create and use bibliographical databases in conjunction with LaTeX, and how to generate typeset bibliographies according to publishers' or style guide expectations.

Chapter 16 describes LaTeX's support for the different citation systems for bibliographical references in common use and how to produce multiple bibliographies by chapter and topic.

Chapter 17 shows how to document LaTeX packages and classes and how to use such files provided by others. It also covers setting up a development and testing environment and working with version control, which is useful for essentially every project.

Appendix A reviews how to handle and manipulate the basic LaTeX programming structures and how to produce class and package files.

Appendix B discusses how to trace and resolve problems and explains common error and warning messages and their likely causes.

Appendix C shows where to go beyond this book if that is ever needed, e.g., how to obtain the packages and systems described, how to access help or take an online course, and much more.

Some of the material covered in the book may be considered "low-level" TEX that has no place in a book about LATEX. However, to the authors' knowledge, much of this information has never been described in the "LATEX" context even though it is important. Moreover, we do not think that it would be helpful simply to direct readers to books like *The TEXbook*, because most of the advice given in books about "plain TEX" either is not directly applicable to LATEX or, worse, produces subtle errors if used with LATEX. In some sections we have, therefore, tried to make the treatment as self-contained as possible by providing all the information about the underlying TEX engine that is relevant and useful within the LATEX context.

1.3.2 Typographic conventions

It is essential that the presentation of the material immediately conveys its function in the framework of the text. Therefore, we present below the typographic conventions used in this book.

Commands, environments, packages, ...

Throughout the text, LATEX command and environment names are set in monospaced type (e.g., `\caption`, `enumerate`, `\begin{tabular}`), while names of packages, class files, and programs are in sans serif type (e.g., article). Commands to be typed by the user on a computer terminal are shown in monospaced type and are underlined, e.g., showing how to call the LATEX development format on the command line:

pdflatex-dev ⟨*file*⟩

Syntax descriptions

The syntax of the more complex LATEX commands is presented inside a rectangular box. Command arguments are shown in italic type:

`\titlespacing*{`*cmd*`}{`*left-sep*`}{`*before-sep*`}{`*after-sep*`}[`*right-sep*`]`

In LATEX, optional arguments are denoted with square brackets, and the star indicates a variant form (i.e., is also optional), so the above box means that the `\titlespacing` command can come in four different incarnations:

`\titlespacing{`*cmd*`}{`*left-sep*`}{`*before-sep*`}{`*after-sep*`}`
`\titlespacing{`*cmd*`}{`*left-sep*`}{`*before-sep*`}{`*after-sep*`}[`*right-sep*`]`
`\titlespacing*{`*cmd*`}{`*left-sep*`}{`*before-sep*`}{`*after-sep*`}`
`\titlespacing*{`*cmd*`}{`*left-sep*`}{`*before-sep*`}{`*after-sep*`}[`*right-sep*`]`

For some commands, not all combinations of optional arguments and/or star forms are valid. In that case the valid alternatives either are explained in the text or are explicitly shown together, as, for example, in the case of LATEX's sectioning commands:

`\section*{`*title*`}` `\section[`*toc-entry*`]{`*title*`}`

Here the optional *toc-entry* argument can be present only in the unstarred form; thus, we get the following valid possibilities:

`\section{`*title*`}` `\section*{`*title*`}` `\section[`*toc-entry*`]{`*title*`}`

Code examples . . .

Lines containing examples with LaTeX commands are indented and are typeset in a monospaced type at a size somewhat smaller than that of the main text:

```
\addtocontents{lof}{\protect\addvspace{10pt}}
\addtocontents{lot}{\protect\addvspace{10pt}}
```

. . . with output . . .

However, in the majority of cases we provide complete examples together with the output they produce side by side:

The right column shows the input text to be treated by LaTeX with preamble material shown in blue. In the left column one sees the result after typesetting.

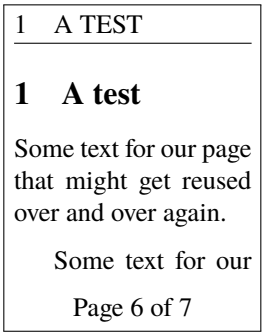
```
\usepackage{ragged2e}
The right column shows the input text to be treated
by \LaTeX{} with preamble material shown in blue.
In the left column one sees the result after
typesetting.
```

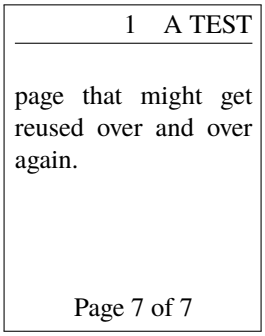
1-3-1

Note that all preamble commands are always shown in blue in the example source.

. . . with several pages . . .

In case several pages need to be shown to prove a particular point, (partial) "page spreads" are displayed and usually framed to indicate that we are showing material from several pages.






```
\usepackage{fancyhdr,lastpage}
\pagestyle{fancy}
\fancyhf{} % --- clear all fields
\fancyhead[RO,LE]{\leftmark}
\fancyfoot[C]{Page \thepage\
              of \pageref{LastPage}}
% \sample defined as before

\section{A test}
\sample \par \sample
```

1-3-2

A number of points should be noted here:

- We usually arrange the examples to show pages 6 and 7 so that a double spread is displayed.
- We often use the command `\sample` to hold a short piece of text to keep the example code short: the definition for this command is either given as part of the example or, as indicated here, repeated from a previous example — which in this case is simply a lie because `\sample` was not defined earlier. In other examples we make use of lipsum or kantlipsum to generate sample text.
- The output may or may not show a header and footer. In the above case it shows both. Because the "pages" are very small but show the real output from the given input on the right, there are often deficiencies in line breaking, etc.

For large examples, where the input and output cannot be shown conveniently alongside each other, the following layout is used:

… with large output …

```
\usepackage{ragged2e,kantlipsum}  \RaggedRight
This is a wide line, whose input commands and output result cannot
be shown nicely in two columns. \kant[1][1-3]
```

Depending on the example content, some additional explanation might appear between input and output (as in this case). Then the output is displayed:

This is a wide line, whose input commands and output result cannot be shown nicely in two columns. As any dedicated reader can clearly see, the Ideal of practical reason is a representation of, as far as I know, the things in themselves; as I have shown elsewhere, the phenomena should only be used as a canon for our understanding. The paralogisms of practical reason are what first give rise to the architectonic of practical reason. As will easily be shown in the next section, reason would thereby be made to contradict, in view of these considerations, the Ideal of practical reason, yet the manifold depends on the phenomena.

1-3-3

Chapter 11 shows yet another example format, where the margins of the example are explicitly indicated with thin blue vertical rules. This is done to better show the precise placement of displayed formulas and their tags in relation to the text margins.

… or with lines indicating the margins

1-3-4

(1) $(a+b)^2 = a^2+2ab+b^2$

```
\usepackage[leqno]{amsmath}
\begin{equation} (a+b)^2 = a^2+2ab+b^2 \end{equation}
```

Some examples make use of color commands, e.g., `\color` or `\textcolor`, but because the book is printed only with two colors, it is not possible to do them justice. The approach we took is that all colors appear as shades of gray except for `blue`, which we changed to produce the "lightblue" that is used as a second color in the book. Thus, all examples actually deploy the declarations as shown in the next example if they use color, but to save space none of them is shown elsewhere.

Color usage in this book

1-3-5

Black blue
red green
yellow blue
bluish

```
\usepackage{xcolor}
\definecolor{blue}{cmyk}{1,0.56,0,0}  % what we call 'blue' in this book
\definecolor{red}{gray}{.7}     \definecolor{green}{gray}{.8}
\definecolor{yellow}{gray}{.9}
Black \textcolor{blue}{blue} \textcolor{red}{red} {\color{green} green}
\textcolor{yellow}{yellow} \colorbox{black!30}{\color{blue} blue}
\fcolorbox{blue}{blue!8}{\color{blue}bluish}
```

The notation `blue!8` is a short form for writing `blue!8!white`. It is xcolor's way to specify simple color mixes and means that we mix 8% blue with 92% white.

All of these examples are "complete" if you mentally add a `\documentclass` line (with the article class[1] as an argument) and surround the body of the example with a `document` environment. In fact, this is how all of the examples in this book were produced. When processing the book, special LaTeX commands take the source lines for an example and write them to an external file, thereby automatically adding the `\documentclass` and the `document` environment lines. This turns each example into a small but complete LaTeX document. These documents are then externally processed (using a mechanism that runs each example as often as necessary, including the generation of a bibliography through BibTeX). The resulting PDF (Portable Document Format) is then cropped to the smallest size that shows all output, using the program pdfcrop and if necessary separated into individual pages using pdfseparate. The resulting graphic files are then loaded in the appropriate place the next time LaTeX is run on the whole book. More details on the actual implementation of this scheme can be found in Section 4.2.4 on page 315.

Watch out for these

Throughout the book, blue notes are sprinkled in the margin to help you easily find certain information that would otherwise be hard to locate. In a few cases these notes exhibit a warning sign, indicating that you should probably read this information even if you are otherwise only skimming through the particular section.

Information relevant only to Unicode TeX engines

Most of the material presented in this book is applicable to all TeX engine flavors, e.g., pdfTeX, XeTeX, or LuaTeX. However, some aspects are applicable only to Unicode engines, and to help you identify this at a glance we have placed such information into boxes like this:

Unicode engines

This is information that applies only to Unicode engines, e.g., XeTeX or LuaTeX.

The only exceptions are Section 9.6 on fontspec and Section 12.4 on unicode-math, both of which would have ended up completely within such boxes — which would be rather hard to read.

Information specific to biblatex/biber

A similar approach is used to highlight any differences between a workflow that uses BibTeX and traditional citation methods and one that uses the biblatex package and the biber program. As both methods have a large overlap, they are described together, and specific considerations are placed into boxes like this:

biber/biblatex

This is information specific to biblatex/biber and often gives tips how to ensure compatibility between the biber/biblatex and the BibTeX workflow.

This convention is used in Chapter 15.

1.3.3 Using the examples

Our aim when producing this book was to make it as useful as possible for our readers. For this reason the book contains more than 1 500 complete, self-contained examples of all aspects of typesetting covered in the book.

[1] Except for examples involving the `\chapter` command, which need the report or book class.

All examples are made available in source format on CTAN at `https://ctan.org/pkg/tlc3-examples`. The examples are numbered per section, and each number is shown in a small box in the inner margin (e.g., 1-3-6 below). These numbers are also used for the external file names by appending `.ltx` (single-page examples) or `.ltx2` (double-page examples).

To reuse any of the examples it is usually sufficient to copy the preamble code (typeset in blue) into the preamble of your document and, if necessary, adjust the document text as shown. In some cases it might be more convenient to place the preamble code into your own package (or class file), thus allowing you to load this package in multiple documents using `\usepackage`. If you want to do the latter, there are two points to observe:

- Any use of `\usepackage` in the preamble code needs to be replaced by a `\RequirePackage` declaration, which is the equivalent command for use in package and class files (see Section A.6.7).
- Any occurrence of `\makeatletter` and `\makeatother` *must* be removed from the preamble code. This is very important because the `\makeatother` would stop correct reading of such a file.

So let us assume you wish to reuse the code from the following example:

A line of text[1] with some[2] footnotes.

1. The first
2. The second

1-3-6

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

You have two alternatives: you can copy the preamble code (i.e., the code colored blue) into your own document preamble or you can place that code — but without the `\makeatletter` and `\makeatother` — in a package file (e.g., `lowfnnum.sty`) and afterwards load this "package" in the preamble of your own documents with `\usepackage{lowfnnum}`.

This page intentionally left blank

CHAPTER 2

The Structure of a LaTeX Document

One of the ideas behind LaTeX is the separation between layout and structure (as far as possible), which allows the user to concentrate on content rather than having to worry about layout issues [106]. This chapter explains how this general principle is implemented in LaTeX.

The first section of this chapter shows how document class files, packages, options, and preamble commands can affect the structure and layout of a document.

The logical subdivisions of a document are then discussed in general, before explaining in more detail how sectioning commands and their arguments define a hierarchical structure, how they generate numbers for titles, and how they produce running heads and feet. This is followed by discussing a few useful packages that allow you to customize different aspects of the layout of sectional units or to provide your own definitions.

In Section 2.3 we take a closer look at the design of table of contents structures and how it can be influenced or extended.

This is followed by a section discussing important packages that support you in providing cross-references that remain correct, even if you change parts of your document. These packages can automatically insert appropriate phrases (varioref, cleveref, nameref), can help you manage your label keys (showkeys and refcheck), or support you in providing references to external documents (xr) or hyperlinks in general (hyperref).

The final section introduces packages and programs that support you in archiving documents or managing them when you work jointly with others on some document.

2.1 The overall structure of a source file

You can use LaTeX for several purposes, such as writing an article or a book or producing presentations. Clearly, documents for different purposes may need different logical structures, i.e., different commands and environments. We say that a document belongs to a *class* of documents having the same general structure (but not necessarily the same typographical appearance). You specify the class to which your document belongs by starting your LaTeX file with a `\documentclass` command, where the mandatory parameter specifies the *name* of the *document class*. The document class defines the available logical commands and environments (for example, `\chapter` in the report class) as well as a default formatting for those elements. An optional argument allows you to modify the formatting of those elements by supplying a list of *class options*. For example, `11pt` is an option recognized by most document classes that instructs LaTeX to choose eleven point as the basic document type size.

Many LaTeX commands described in this book are not specific to a single class but can be used with several classes. A collection of such commands is called a *package*, and you inform LaTeX about your use of certain packages in the document by placing one or more `\usepackage` commands after `\documentclass`.

Just like the `\documentclass` declaration, `\usepackage` has a mandatory argument consisting of the *name* of the package and an optional argument that can contain a list of *package options* that modify the behavior of the package.[1]

The document classes and the packages reside in external files with the extensions `.cls` and `.sty`, respectively. Code for options is sometimes stored in external files (in the case of class files with the extension `.clo`) but is normally directly specified in the class or package file (see Appendix A for information on declaring options in classes and packages). However, in the case of options, the file name can differ from the option name. For example, the option `11pt` is related to `size11.clo` when used in the article class and to `bk11.clo` inside the book class.

The document preamble

Commands placed between `\documentclass` and `\begin{document}` are in the so-called *document preamble*. All style parameters must be defined in this preamble, either in package or class files or directly in the document *before* the `\begin{document}` command, which sets the values for some of the global parameters. A typical document preamble could look similar to the following:

```
\documentclass[twocolumn,a4paper]{article}
\usepackage{multicol}
\usepackage[ngerman,french]{babel}
\addtolength\textheight{3\baselineskip}
\begin{document}
```

This document preamble defines that the class of the document is article and that the layout is influenced by the formatting request `twocolumn` (typeset in two columns) and the option `a4paper` (print on A4 paper). The first `\usepackage` declaration

[1] These commands also have a second optional argument that is intended for cases where a specific release of a package or a document class is required. This is discussed in Section 2.5.5 on page 114.

informs LaTeX that this document contains commands and structures provided by the package multicol. In addition, the babel package with the options `ngerman` (support for German language) and `french` (support for French language) is loaded. Finally, the default height of the text body was enlarged by three lines for this document.

Generally, nonstandard LaTeX package files contain modifications, extensions, or improvements[1] with respect to standard LaTeX, while commands in the preamble define changes for the current document. Thus, to modify the layout of a document, you have several possibilities:

- Change the standard settings for parameters in a class file with options defined for that class.
- Add one or more packages to your document and make use of them.
- Change the standard settings for parameters in a package file with options defined for that package.
- Write your own local packages containing special parameter settings and load them with `\usepackage` after the package or class they are supposed to modify (as explained in the next section).
- Make final adjustments inside the preamble.

If you want to get deeper into LaTeX's internals, you can, of course, define your own general-purpose packages that can be manipulated with options. You find additional information on this topic in Appendix A.

2.1.1 Spoiler alert — The `\DocumentMetadata` command

When LaTeX changed from LaTeX 2.09 to LaTeX 2_ε around 1994, the overall document structure was slightly changed to automatically distinguish old from new documents (to switch to compatibility mode, if necessary). LaTeX 2_ε documents start with `\documentclass` as described above, while LaTeX 2.09 documents started with the command `\documentstyle`, and `\usepackage` was unavailable.

Now, roughly a quarter century later, there is another major shift under way during which LaTeX is being modernized to support accessible PDF/UA (Portable Document Format/Universal Accessibility) and other functionality that is important for it to remain useful; see the discussion in Section 1.1 on page 7. This time around, the functionality change is essentially upward compatible, and old documents can be easily reprocessed using the new features. Thus, instead of dividing documents into two classes (old and new) by changing the first command, you can now indicate that you want to use the new functionality by adding a `\DocumentMetadata` declaration in front of `\documentclass` while leaving the rest of the document unchanged.

[1] Many of these packages have become de facto standards and are described in this book. This does not mean, however, that packages that are not described here are necessarily less important or useful, of inferior quality, or should not be used. We merely concentrated on a few of the more established ones; for others, we chose to explain what functionality is possible in a given area.

`\DocumentMetadata{`*key/value list*`}`

This declaration should be the first command in a document; i.e., if present, it should come before `\documentclass`. It expects a *key/value list* as its argument in which you specify "metadata" about the document that guides the production of the final output, e.g., should it adhere to a certain standard, should it be a tagged PDF, what is its author, title, and keywords that are shown in the metadata of the resulting PDF, etc. All these "metadata" are stored so that packages and users can access the data in a consistent way.

For example, the key `pdfversion` allows you to set the PDF version. With the key `pdfstandard` it is possible to require a standard such as `A-2b`. If that is specified, it directs LaTeX to embed an appropriate color profile and set up verification tests that packages like hyperref can use to suppress actions not allowed in this standard. A further example is the `backend` key that allows you to specify a backend, e.g., `dvipdfmx` or `dvisvg`, which is useful in cases where the correct backend cannot be detected automatically.

At the time of writing this book the details about which other keys are going to be supported are still open (the whole exercise is a multi-year project [150] after all), but what we can say is that already now you can use this future interface to enable some new functionality. For example, just adding

```
\DocumentMetadata{}
\documentclass{article}        % (or any other class)
...                            % with preamble as previously
\begin{document}
```

is enough to load the new support code for managing PDF output, and this enables packages, such as hyperref, to provide features otherwise not available; see Section 2.4.6 on page 96 for details.

2.1.2 Processing of options of the document class and packages

You can think of options to the document class or to packages as a simple way to adjust some of the properties of the whole document (when used in `\documentclass`) or of properties of individual packages (if specified in `\usepackage`). More fine-grain control is usually also possible through declarations and setup commands that are defined by a class or package file and are available for use once that file is loaded.

You can specify options in a `\usepackage` command only if these options are explicitly declared by the package. Otherwise, you receive an error message, informing you that your specified option is unknown to the package in question. Options to the `\documentclass` are handled slightly differently. If a specified option is not declared by the class, it is assumed to be a "global option".

All options given to `\documentclass` (whether declared or global) are automatically passed as class options to all `\usepackage` declarations. Thus, if a package file loaded with a `\usepackage` declaration recognizes (i.e., declares) some of the class options, it can take appropriate actions. If not, the class options are ignored while processing that package. Because all options have to be defined inside the class or package file, their actions are under the control of the class or package (an action

can be anything from setting internal switches to reading an external file). For this reason their order in the optional argument of `\documentclass` or `\usepackage` is (usually) irrelevant.

If you want to use several packages, all taking the same set of options (for example, none), it is possible to load them all with a single `\usepackage` command by specifying the package names as a comma-separated list in the mandatory argument. For example,

```
\usepackage[ngerman]{babel}   \usepackage[ngerman]{varioref}
\usepackage{array}            \usepackage{multicol}
```

is equivalent to

```
\usepackage[ngerman]{babel,varioref} \usepackage{array,multicol}
```

By specifying `ngerman` as a global option to the class we can further shorten the `\usepackage` declaration as `ngerman` is passed to all loaded packages and thus will be processed by those packages that declare it.

```
\documentclass[ngerman]{book}
\usepackage{babel,varioref,array,multicol}
```

Of course, this assumes that neither array nor multicol changes its behavior when `ngerman` is passed as a class option.

Finally, when the `\begin{document}` is reached, all global options are checked to see whether each has been used by at least one package; if not, a warning message is displayed. It is usually a spelling mistake if your option name is never used; another possibility is the removal of a `\usepackage` command loading a package that used this option previously.

Key/value options and their limitations

When the option concept was originally developed, it was based on the idea that options are simple strings separated by commas without further structure. Spaces in that option list are explicitly ignored, because people often split such option lists over several lines and inadvertently introduced spaces before or after the commas. After a while some package developers started to use a key/value concept for options or setup commands; e.g., `geometry` allows you to write `paper=a4,margin=1in` with the meaning that the option `paper` gets the value `a4` and `margin` is set to one inch. That works if neither the option name nor the intended value requires spaces because those get stripped away if used in a class or package option list.[1]

This limitation is not easy to overcome for existing implementations without huge backward compatibility issues, which means that it is usually best to use a setup command (if provided by a package) rather than the option list with such packages, because in a setup command spaces are honored except those next to commas and equal signs. With the new key/value methods directly supported by the LaTeX format, spaces are trimmed only at either end (where one would expect it). For new packages or package reimplementations we therefore recommend using LaTeX's mechanism, which is described in Appendix A.6.6 on page →II 700.

Configuration after loading a package

If you want to make some modifications to a document class or a package (for example, changing parameter values or redefining some commands), you can put the relevant code into a separate file with the extension `.sty`. Then load this file with a

[1] This restriction is lifted in very new packages using the L3 programming layer methods.

\usepackage command after the package whose behavior you wish to modify (or the document class, if your modifications concern class issues).

Alternatively, you can insert the modifications directly into the preamble of your document. In that case, you may have to bracket them with \makeatletter and \makeatother if they contain internal LaTeX 2ε commands (i.e., those with an @ sign in their names) or use \ExplSyntaxOn and \ExplSyntaxOff if they are LaTeX3 commands (i.e., with _ and : in their names). For more details see the discussion on page →II 623 concerning internal commands in the preamble.

2.1.3 Front, main, and back matter

In a longer document, such as a book or a longer article, we usually can identify three distinct areas: the front matter, the main matter (or body matter), and the back matter.

As the name indicates the main matter holds the main text, while the two other parts provide supplementary information before and after. The front matter typically consists of the title page or pages, the table of contents and similar lists, an abstract, and a foreword or preface (though the latter may already be thought of belonging to the main matter). To the back matter you typically count any appendices, bibliography, index, and afterword, colophon, etc.

Typographically these three regions are often handled in different ways to make them easily identifiable, for example, by using different page numbering systems for front and main matter[1], not numbering headings in the front matter, and often using different heading number styles in main and back matter.

In shorter works this distinction becomes somewhat blurry: the front matter may just consist of the title (and not even on a page of its own) in which case it makes more sense to think of it as belonging to the main matter. Similarly, even in longer works there may not be any back matter.

In LaTeX's book class these three regions can be explicitly marked up using the commands \frontmatter, \mainmatter, and \backmatter. In other classes you often find only the command \appendix, which is used to separate the body matter from the back matter — the assumption being that in articles and similar documents the front matter due to its length does not require special typographical treatment.

Front matter elements

The standard LaTeX classes provide \title, \author (with \and and \thanks) and \date to set up the title information and \maketitle to produce the actual document title. For more elaborate title pages they offer the environment titlepage, which basically gives you an empty page in which you have to draw and position your title yourself.[2]

[1] If you prefer the front and main matter to use the same page numbering system, check out the little package arabicfront by Javier Bezos. It works with most document classes and results in the front and main matter being numbered with arabic numerals in a continuing sequence.

[2] Please note that in many classes the titlepage environment sets the page number explicitly to one and then issues a \thispagestyle{empty} to hide it. The downside is that this looks internally to LaTeX always like a recto page, which in a twoside setting might cause problems. Thus, even though

Producing title pages

If you design your own title page, it might be worth taking a look at the collection of title page examples by Peter Wilson [199], which contains forty examples together with the (sometimes low-level) code to produce them. Another possibly helpful resource is the package titling by the same author, which provides methods for restyling the material produced by `\title`, `\author`, `\thanks`, `\date`, and `\maketitle`.

The support offered by the standard classes (article, report, or book) is not really sufficient for anything other than preprints, which is why classes for specific journals or classes targeting book production often offer additional commands for specifying data relevant for the title or even provide some totally different commands altogether. This is an area where, due to the lack of decent support in the standard classes, the document syntax unfortunately varies from class to class, so you have to consult the appropriate documentation to see what is necessary for a particular class.

Complex author information

A possible alternative is the little package authblk by Patrick Daly that provides an extended syntax for the `\author` command and can typeset affiliation information either in blocks (below each group of authors) or as footnotes as shown in the next example. By using an optional argument to `\author` and/or to `\affil`, it is even possible to have author and affiliations ordered in different ways. The package offers a number of customization possibilities, two of which are shown in the example; consult the documentation for further details. It should work with most document classes even if they provide their own author management.

Author Management

Immanuel Kant[1], Moses Mendelssohn[2], Friedrich Schiller[3], Leonhard Euler[4], and Friedrich der Grosse*[2]

[1]*Königsberg*
[2]*Berlin*
[3]*Jena*
[4]*St. Petersburg*

June, 1770

As any dedicated reader can clearly see, the Ideal of practical reason is a representation of, as far as I know, the things in themselves; as I have shown elsewhere, the phenomena should

2-1-1

*Sponsor

```
\usepackage[auth-sc,affil-it]
           {authblk}
\usepackage{kantlipsum}

\title{Author Management}

\author{Immanuel Kant}
  \affil{Königsberg}
\author{Moses Mendelssohn}
  \affil{Berlin}
\author{Friedrich Schiller}
  \affil{Jena}
\author{Leonhard Euler}
  \affil{St.\ Petersburg}
\author[2]{Friedrich der
           Große\thanks{Sponsor}}

\date{June, 1770}

\maketitle
\kant[1]   % only partly shown
```

the page number is suppressed, you may have to adjust the page number to a different number inside (and again afterwards) if the page is meant to be a verso page.

Various content lists

For the typical lists found in the front matter, such as the table of contents, the standard classes support the commands `\tableofcontents`, `\listoftables`, and `\listoffigures`. Additional lists can be defined as explained in Section 2.3.4 on page 74. Typically such lists produce unnumbered headings. If your front matter requires further sectional units, such as a foreword or a preface, produce them with the star form of a suitable heading command, e.g., `\chapter*` or `\section*`.

Abstracts

Another important element, in particular for articles, is the `abstract` environment. Note that unfortunately the correct placement of this environment may depend on the chosen document class. In the standard classes and many others it is typeset where specified in the source, but there are classes in which it is formatted and placed by the `\maketitle` command and therefore has to appear before it. Its default formatting is usually adequate, and if you are typesetting an article for some particular journal, you should probably not alter it. However, if you do not like the outcome and you are free to make changes, take a look at the abstract package by Peter Wilson that offers a large arsenal of bells and whistles for adjusting most aspects of the abstract layout.

Other nonstandard elements

There are other important frontmatter elements, such as a keyword list in journal articles, or bibliographic and copyright information in books, but none of these is provided for by the standard classes. However, in document classes for specific journals or book series from publishers, you usually find additional commands and environments that cater for these elements. Typically they differ from class to class so that one has to redo this part of the frontmatter if the document class is changed.

Main matter elements

The top-level structural elements of the body text are various levels of heading commands that are discussed in detail in Section 2.2 on page 32 and of course lists and other elements discussed in Chapter 4.

Back matter elements

Probably the most often used back matter elements are a bibliography and an index, which are supported through the environments `theindex` and `thebibliography` discussed in more detail in Chapters 14 and 15.

If you have several other appendices, use heading commands of the appropriate level to introduce them. The numbering scheme for such headings is automatically adjusted by the `\appendix` or `\backmatter` declaration that separates the back matter material from the main text. However, if there is only a single appendix, it may look odd if that gets numbered. Thus, in that case, you may explicitly want to use the star form of the heading command.

2.1.4 Splitting the source document into several files

LaTeX source documents can be conveniently split into several files by using `\input` or `\include` commands. The `\input` command unconditionally includes the file specified as its argument at the current point. This is useful if you want to split your

document into reasonably sized chunks or you want to reuse some parts for one or the other reason and therefore want to keep them in separate files.[1]

The `\include` command, however, is different in that it automatically starts a new page before and after the included file. For each `\include` file a separate `.aux` file is produced, which is why in contrast to `\input` such files should be specified without extension and on the operating system level always have the extension `.tex`.

`\include` used without extension

The reason for `\include` is that documents can be reformatted piecewise by specifying as arguments of an `\includeonly` declaration only those `\include` files LaTeX has to reprocess. For the other files that are loaded with `\include` commands, the counter information (page, chapter, table, figure, equation, ...) is then read from the corresponding `.aux` files generated during a previous run. In the following example, the user wants to reprocess only the files `chap1.tex` and `appen1.tex`:

Partial processing

```
\documentclass{book}       % the document class ‘‘book’’
\includeonly{chap1,appen1} % only include chap1 and appen1
\begin{document}
\include{chap1}            % input chap1.tex
\include{chap2}            % input chap2.tex
  ...                      % ... further chapters
\include{appen1}           % input appen1.tex
\include{appen2}           % input appen2.tex
\end{document}
```

Be aware that LaTeX issues only a warning message like `"No file xxx.tex"` and not an error message when it cannot find a file specified in an `\include` statement and then continues processing.

If the information in the `.aux` files is up-to-date, it is possible to process only part of a document and have all counters, cross-references, and pages be correct in the reformatted part. However, if one of the counters (including the page number for cross-references) changes in the reprocessed part, then the complete document might have to be rerun to get the index, table of contents, and bibliographic references consistently correct.

Note that each document part loaded via `\include` starts on a new page and finishes by calling `\clearpage`; thus, floats contained therein do not move outside the pages produced by this part. Natural candidates for `\include` are therefore whole chapters of a book but not necessarily small fractions of text.

While it is certainly an advantage to split a larger document into smaller parts and to work on more manageable files with a text editor, partial reformatting should be used only with great care and when still in the developing stage for one or more chapters. When a final and completely correct copy is needed, the only really safe procedure is to reprocess the complete document. However, if the document is too large to process in a single run, make sure that for the final version the pieces are processed *in the correct sequence* (if necessary several times) to ensure that the cross-references and page numbers are correct.

Avoid using partial processing when preparing the final version of your document

[1] Not everything can be placed into separate `\input` files, though. For example, it is not possible to put only a part of a `tabular` environment in a file; it has to go in completely.

Some packages are incompatible with the `\include` *mechanism*

It is very important to note that some packages can not be used reliably with the `\include` mechanism. Likely candidates are those that write their own support files to store data between runs as they often do not realize that parts of the document are not processed. A premier example from this book is the acro package. It always considers the first acronym it sees as being the acronym that is showing the full form; thus, if you apply `\includeonly`, it may see different instances as being the first, thereby altering the line breaking and pagination compared to always processing the full document.

2.1.5 askinclude — Managing your inclusions

Interactive inclusion

If you intend to work with `\include` commands, consider using the small package askinclude created by Pablo Straub and Heiko Oberdiek. It interactively asks you which files to include. You can then specify the files as a comma-separated list (i.e., what you would put into the `\includeonly` argument) or use `*` to indicate all files, `-` to include no files or `?` in which case it asks you for each include file separately. Alternatively, if the Enter button is pressed in response, then your answer from the previous run is used again. This way you do not have to modify your master source to process different parts of your document (a very useful feature during the production of this book). All this works by storing the answer given in the `.aux` file so that it is available again on the next run. Thus, if that file is removed for some reason, you have make your selection again and cannot simply hit Enter.

The package also offers some pattern matching facilities if enabled with the option `makematch`. In this case `*` matches zero or more arbitrary characters, and a `!` at the start of a pattern negates its effect (i.e., excludes matching names). For example, `chap*,!chap1` would include all files starting with `chap` except `chap1`.

2.1.6 tagging — Providing variants in the document source

Sometimes it is useful to keep several versions of a document together in a single source, especially if most of the text is shared between versions. This functionality is provided by the tagging package[1] created by Brent Longborough (1944–2021).

`\tagged{`*label-list*`}{`*text*`}`	`\usetag{`*label-list*`}`	`\droptag{`*label-list*`}`

The variant text parts are specially marked in the source using the command `\tagged`, and during formatting some of them are selected. The command takes two arguments: a label (or a comma-separated list of labels) that describes to which variant the optional text belongs, and the text to be conditionally printed.

With the command `\usetag` in the document preamble you can select which label (or labels) is active at the beginning of the document. Alternatively, you can specify the labels as package options to activate them. Inside the document body you

[1] A number of other packages provide similar functionality with slightly different interfaces, e.g., comment by Victor Eijkhout, xcomment by Timothy Van Zandt, and optional and version by Donald Arseneau. There is also multiaudience by Boris Veytsman, which uses a quite different approach that might be more suitable in complex situations.

can use further \usetag commands to activate additional labels, and you can use \droptag to inactivate some of them.

\untagged{*label-list*}{*text*}	\iftagged{*label-list*}{*yes-test*}{*no-text*}

For convenience there is also \untagged, which typesets its second argument if none of its labels is currently active. Finally, there is \iftagged with three arguments that print the second or third argument depending on the given labels in the first.

All five commands are shown in the following example:

2-1-2

```
\usepackage[doc]{tagging}
\tagged{doc}   {Typeset this if tag doc is used.}
\untagged{code}{Typeset this if tag code is not used.}
\iftagged{be}  {To be or}{Not to be!}  Which is it?   \par
\tagged{doc,code}{Typeset this for either doc or code.}
Typeset this \untagged{}{always}\tagged{}{never}!     \par
\usetag{code} \droptag{doc}
Now neither of the variants are typeset!
\tagged{doc}   {Typeset this if tag doc is used.}
\untagged{code}{Typeset this if tag code is not used.}
```

Typeset this if tag doc is used. Typeset this if tag code is not used. Not to be! Which is it?

Typeset this for either doc or code. Typeset this always!

Now neither of the variants are typeset!

This approach works well enough for shorter texts but has the limitation that it cannot contain \verb commands and must have balanced braces because the *text* is provided as an argument. With longer parts to be optionally printed, however, it is usually best to either store them in an external file and conditionally load this file in a \tagged command or use the environments shown in the next example.

2-1-3

```
\usepackage[doc]{tagging}
Environments can contain verbatim material
\begin{taggedblock}{doc} e.g., \verb=#&=\end{taggedblock}
.  \par Note the placement of the period and the spacing!
Careful: \begin{untaggedblock}{doc}
                                Not \end{untaggedblock} shown!
```

Environments can contain verbatim material e.g., #&.

Note the placement of the period and the spacing! Careful:

Please note the surprising placement of the period. You should never place anything after the \end{taggedblock} or \end{untaggedblock}, because it gets discarded if the environment body is not typeset. This can be seen by the missing word “shown!” in the result. This may not be immediately apparent, because as long as the optional material is typeset, everything appears to be fine, but the moment the material is ignored, the rest of the last line vanishes too. Best practice is therefore to place the \begin and \end commands on lines by themselves.

The handling of space is also a bit peculiar: inside the environment body spaces are honored, except for spaces immediately following the \begin command. This is why we do not see two spaces in the output but only one, even though there is a space before and after \begin. If we had added a space before the \end command, it would have resulted in “# .” in the output.

\part	top-level	(level −1 in book and report; level 0 in article)	
\chapter	level 0	(only defined by book and report)	
\section	level 1		
\subsection	level 2	\paragraph	level 4
\subsubsection	level 3	\subparagraph	level 5

Table 2.1: LaTeX's standard sectioning commands

The tagging package selects the variants to process during the LaTeX formatting. Depending on the application, it might be better to use a different approach involving a preprocessor that extracts individual variants from the master source. For example, the docstrip program can be successfully used for this purpose; in contrast to other preprocessors, it has the advantage that it is usable at every site that has an installed LaTeX system (see Section 17.2 for details).

2.2 Sectioning commands

In the previous section we discussed the top-level division into front, main, and back matter. Within these regions further division is done through sectional units that are typically substructured. These we discuss in this section.

The standard LaTeX document classes (i.e., article, report, and book) contain commands to define the different hierarchical structural units of a document (e.g., chapters, sections, subsections, etc.). Each such command defines a nesting level inside a hierarchy, and each structural unit belongs to some level. The commands should be correctly nested. For example, a \subsection command should be issued only after a previous \section.

Standard LaTeX provides the set of sectioning commands[1] shown in Table 2.1. The \chapter command defines level zero of the hierarchical structure of a document, \section defines level one, and so on, whereas the optional \part command defines the level minus one (or zero in classes that do not define \chapter). Not all of these commands are defined in all document classes. The article class does not have \chapter, and the letter class does not support sectioning commands at all. It is also possible for a package to define other sectioning commands, allowing either additional levels or variants for already supported levels.

The standard names are admittedly somewhat strange; e.g., \paragraph does not mean as one might expect "start a new text paragraph" but instead "here is the heading for the next subsubsubsection". So if you prefer a different name for such units in your documents, a definition such as

```
\newcommand\subsubsubsection{\paragraph}
```

[1] Using commands instead of environments to indicate the sectional units has the effect that these heading commands do not define a scope; e.g., parameter changes stay in force across different sectional units.

would easily fix that (though is that name really better?). Note that it only means your document uses a different command: the actual work is still carried out by `\paragraph`, and the counter associated with the unit is still called `paragraph` and printed with `\theparagraph` and so on.

`\section[`*toc-entry*`]{`*title*`}` `\section*{`*title*`}`

All standard sectioning commands — i.e., `\part`, `\chapter` (only in the `book` and `report` classes), `\section`, `\subsection`, `\subsubsection`, `\paragraph`, and `\subparagraph` — have a common syntax as exemplified here by the `\section` command. Generally, the sectioning commands automatically perform one or more of the following typesetting actions:

- produce the heading number reflecting the hierarchical level;
- store the heading as an entry for a table of contents (into the `.toc` file);
- save the contents of the heading to be (perhaps) used in a running header/footer;
- format the heading.

The first form performs all of the above actions. If the optional argument *toc-entry* is present, it is used as the text string for the table of content and the running header and/or footer; otherwise, the *title* is also used for those places. In particular this means that you cannot specify different texts for the table of content and for the running header through this interface. The numbering depends on the current value of the counter `secnumdepth` (discussed in the next section).

Problems with explicit formatting

If you try to advise TEX on how to split the heading over a few lines using the "`~`" symbol or the `\\` command, then side effects may result when formatting the table of contents or generating the running head. In this case the simplest solution is to repeat the heading text without the specific markup in the optional parameter of the sectioning command.

The starred form (e.g., `\section*{...}`) suppresses the numbering for a title and does not produce an entry in the table of contents or the running head. This is usually used inside the front matter and sometimes in the back matter but can, of course, be used anywhere within the document. In the standard classes, the commands `\tableofcontents`, `\listoftables`, and `\listoffigures`, and the `theindex` and `thebibliography` environments internally invoke the command (`\section` or `\chapter`) using their starred form.

The remainder of this section discusses how the appearance of headings can be adjusted to your needs. First we explain how heading numbers work and how they can be manipulated. We then take a quick look at the various fixed texts produced by some headings and how they can be altered. In Sections 2.2.3 to 2.2.7 we describe several packages for heading design, mainly focusing on the `titlesec` package, as that is a good toolbox for most heading design requirements. Finally, we conclude with a discussion of LATEX's low-level interfaces for this area — a section largely meant for reference only (which is why it is set in a smaller font to save space).

2.2.1 Numbering headings

To support numbering, LaTeX uses a counter for each sectional unit and composes the heading number from these counters.

Numbering no headings

Perhaps the change desired most often concerning the numbering of titles is to alter the nesting level up to which a number should be produced. This is controlled by a counter named `secnumdepth`, which holds the highest level with numbered headings. For example, some documents have none of their headings numbered. Instead of always using the starred form of the sectioning commands, it is more convenient to set the counter `secnumdepth` to `-2` in the document preamble. The advantages of this method are that an entry in the table of contents can still be produced and that arguments from the sectioning commands can produce information in running headings. As discussed, these features are suppressed in the starred form.

Numbering all headings

To number all headings down to `\subparagraph` or whatever the deepest sectioning level for the given class is called, setting the counter to a high enough value (e.g., a declaration such as `\setcounter{secnumdepth}{5}` would be sufficient for the standard classes).

Numbering more or less heading levels

Finally, the `\addtocounter` command provides an easy way of numbering more or fewer heading levels without worrying about the level numbers of the corresponding sectioning commands. For example, if you need one more level with numbers, you can place `\addtocounter{secnumdepth}{1}` in the preamble of your document without having to look up the right value. In some cases this might even be useful within the document; see also the package tocvsec2 by Peter Wilson that provides further support for such occasions.

Every sectioning command has an associated counter, which by convention has the same name as the sectioning command (e.g., the command `\subsection` has a corresponding counter `subsection`). This counter stores the current number of sectional units of the level, but its print representation (that you get with `\the`*`counter`*) holds the full formatted number for the given sectioning command. Thus, in the report class, the commands `\chapter`, `\section`, `\subsection`, and so on, represent the hierarchical structure of the document, and a counter like `subsection` keeps track of the number of `\subsections` used inside the current `\section`, e.g., holds the value `1` at this point in the book, while `\thesubsection` would generate `2.2.1`.

Normally, when a counter at a given hierarchical level is incremented, then the next lower-level counter (i.e., that with the next higher-level number) is reset. For example, the report class file contains the following declarations:

```
\newcounter{part}                       % (-1) parts
\newcounter{chapter}                    % (0)  chapters
\newcounter{section}[chapter]           % (1)  sections
\newcounter{subsection}[section]        % (2)  subsections
\newcounter{subsubsection}[subsection]% (3)  subsubsections
\newcounter{paragraph}[subsubsection]  % (4)  paragraphs
\newcounter{subparagraph}[paragraph]   % (5)  subparagraphs
```

These commands declare the various counters. The level one (`section`) counter

is reset when the level zero (chapter) counter is stepped. Similarly, the level two (subsection) counter is reset whenever the level one (section) counter is stepped. The same mechanism is used down to the \subparagraph command. Note that in the standard classes the part counter is decoupled from the other counters and has no influence on the lower-level sectioning commands. As a consequence, \chapters in the book or report class or \sections in article are numbered consecutively even if a \part command intervenes. Changing this inside a class is simple — you just replace the corresponding declaration of the chapter counter with:

```
\newcounter{chapter}[part]
```

The behavior of an already existing counter can be changed with the commands \counterwithin or \counterwithout (see Appendix A.2.1); for example, to alter the behavior for just a single document, you can use

```
\counterwithin{chapter}{part}
```

Every counter in LaTeX, including the sectioning counters, has an associated command constructed by prefixing the counter name with \the, which generates a typeset representation of the counter in question. In the case of the sectioning commands, this representation form is used to produce the full number associated with the commands, as in the following definitions:

```
\renewcommand\thechapter{\arabic{chapter}}
\renewcommand\thesection{\thechapter.\arabic{section}}
\renewcommand\thesubsection{\thesection.\arabic{subsection}}
```

In this example, \thesubsection produces an Arabic number representation of the subsection counter prefixed by the command \thesection and a dot. This kind of recursive definition facilitates modifications to the counter representations because changes do not need to be made in more than one place. If, for example, you want to number sections using capital letters, you can redefine the command \thesection:

```
\renewcommand\thesection{\Alph{section}}
\section{Different-looking section}
\subsection{Different-looking subsection}

Due to the default definitions not only the
numbers on sections change, but lower-level
sectioning commands also show this
representation of the section number.
```

A Different-looking section

A.1 Different-looking subsection

Due to the default definitions not only the numbers on sections change, but lower-level sectioning commands also show this representation of the section number.

2-2-1

Thus, by changing the counter representation commands, it is possible to change the number displayed by a sectioning command. However, the representation of the number cannot be changed arbitrarily by this method. Suppose you want to produce a

subsection heading with the number surrounded by a box. Given the above examples, one straightforward approach would be to redefine `\thesubsection`; e.g.,

```
\renewcommand\thesubsection{\fbox{\thesection.\arabic{subsection}}}
```

But this is not a good approach, as one sees when trying to reference such a section.

3.1 A mistake

Referencing a subsection in this format produces a funny result as we can see looking at subsection 3.1. We get a boxed reference.

```
\renewcommand\thesubsection
    {\fbox{\thesection.\arabic{subsection}}}
\setcounter{section}{3}
\subsection{A mistake}\label{wrong}
Referencing a subsection in this format produces
a funny result as we can see looking at
subsection~\ref{wrong}. We get a boxed reference.
```

2-2-2

In other words, the counter representation commands are also used by LaTeX's cross-referencing mechanism (the `\label` and `\ref` commands; see Section 2.4). Therefore, we can make only small changes to the counter representation commands so that their use in the `\ref` command still makes sense. To produce the box around the heading number without spoiling the output of a `\ref`, we would have to redefine LaTeX's internal command `\@seccntformat`, which is responsible for typesetting the counter part of a section title. As this is rather messy, it is better to use the interface provided by the titlesec package for this, which is what we do in the next example.

1 This is correct

Referencing a section using this definition generates the correct result for the section reference 1.

```
\usepackage{titlesec}
\titlelabel{\fbox{\thetitle}\hspace{0.5em}}
\section{This is correct}\label{sec:OK}
Referencing a section using this definition
generates the correct result for the section
reference~\ref{sec:OK}.
```

2-2-3

The framed box around the number in the section heading is now typeset only as part of the heading, and hence the reference labels come out correctly. Within `\titlelabel` the command `\thetitle` refers to the section counter representation; e.g., it evaluates to `\thesection` in this case. Also note that we reduced the space between the box and the text to 0.5 em (instead of the default 1 em). Another often asked for use case for `\titlelabel` is adding a period after the heading number (but not when referencing it). This is shown in Example 2-2-8 on page 41.

A declaration done with `\titlelabel` applies to all headings. Therefore, if you wish to use different definitions for different headings, you must put the appropriate code into every heading definition instead (which requires the extended interface of titlesec; see page 42).

2.2.2 Changing fixed heading texts

Some of the standard heading commands produce predefined texts. For example, `\chapter` produces the string "Chapter" in front of the user-supplied text. Similarly,

Command	*Default String*	*Command*	*Default String*
\abstractname	Abstract	\indexname	Index
\appendixname	Appendix	\listfigurename	List of Figures
\bibname	Bibliography	\listtablename	List of Tables
\chaptername	Chapter	\partname	Part
\contentsname	Contents	\refname	References

\refname *is used by* article *class;* \bibname *by* report *and* book.

Table 2.2: Language-dependent strings for headings

some environments generate headings with predefined texts. For example, by default the abstract environment displays the word "Abstract" above the text of the abstract supplied by the user. LaTeX defines these strings as command sequences (see Table 2.2) so that you can easily customize them to obtain your favorite names. This is shown in the example below, where the default name "Abstract", as defined in the article class, is replaced by the word "Summary".

2-2-4

Summary

This book describes how to modify the appearance of LaTeX documents.

```
\renewcommand\abstractname{Summary}
\begin{abstract}
 This book describes how to modify the
 appearance of \LaTeX{} documents.
\end{abstract}
```

The standard LaTeX class files define a few more strings. See Section 13.1.3, and especially Table 13.2 on page →II 305, for a full list and a discussion of the babel system, which provides translations of these strings in more than sixty languages.

2.2.3 Introduction to heading design

Headings can be loosely subdivided into two major groups: display and run-in headings. A display heading is separated by a vertical space from the preceding and the following text — most headings in this book are of this type.

A run-in heading is characterized by a vertical separation from the preceding text, but the text following the title continues on the same line as the heading itself, only separated from the latter by a horizontal space. In many classes the lower-level headings such as \paragraph are formatted as run-in headings. Note that an empty line after the heading command is ignored.

2-2-5

Run-in headings. This example shows how a run-in heading looks like. Paragraph text following the heading continues on the same line as the heading.

```
\paragraph{Run-in headings.}

This example shows how a run-in heading looks
like.  Paragraph text following the heading
continues on the same line as the heading.
```

In the remainder of this section we are now going to look at how display and run-in headings can be designed and how one can adjust a given design. We start by looking at two packages that offer a somewhat special feature: they add quotations to display headings.

We then discuss all other design aspects that are supported through the high-level interfaces of the titlesec package. At the very end we also briefly look at the low-level support offered by LaTeX because this is helpful in understanding the code found in older document class files.

2.2.4 quotchap, epigraph — Mottos on chapters and sections

An interesting way to enhance `\chapter` headings is provided by the `quotchap` package created by Karsten Tinnefeld with later updates by Jan Klever. It allows the user to specify quotation(s) that will appear on the top left of the chapter title area.

The quotation(s) for the next chapter are specified in a `savequote` environment; the width of the quotation area can be given as an optional argument defaulting to `10cm`. Each quotation should finish with a `\qauthor` command to denote its source, though it would be possible to provide your own formatting manually.

The default layout produced by the package can be described as follows: the quotations are typeset in `\slshape`, placed flush left, followed by vertical material stored in the command `\chapterheadstartvskip`. It is followed by a very large chapter number, typeset flush right in 60% gray, followed by the chapter title text, also typeset flush right. After a further vertical separation, taken from the command `\chapterheadendvskip`, the first paragraph of the chapter is started without indentation.

The number can be printed in black by specifying the option `nogrey` to the package. To print the chapter number in one of the many freely available fonts, you can choose among a dozen of options, such as `charter` for Bitstream's Charter BT or `times` for Adobe's Times. By default, Adobe's Bookman is chosen. Alternatively, you can explicitly specify a font family (basically any of those listed in the tables in Chapter 10) as an argument to `\qsetcnfont`. Or you could redefine the `\chapnumfont` command, which is ultimately responsible for selecting the font and font size for the chapter number.

The `\quotefont` command defines the font used for the quote, and with the help of `\qauthorfont` you can alter the font for the author name (which is why we still get a sans serif font in the example even though only `\scshape` was specified). Finally, the font for the chapter title font can be influenced by redefining the `\sectfont` command as shown in the example.

This, together with the possibilities offered by redefining the commands `\chapterheadstartvskip` and `\chapterheadendvskip`, allows you to produce a number of interesting layouts even though a lot remains hardwired.[1] The following example uses a negative vertical skip to move the quotation on the same level as the number (in Avantgarde) and set the title and quotation in Helvetica (or more exactly in TeX Gyre Heros).

[1] If you require more customization, you have to define your own variation of the command `\@makechapterhead` starting from the code found in the package.

Cookies! Give me some cookies!
Cookie Monster

1 A Package Test

```
\usepackage[avantgarde]{quotchap}
\renewcommand\chapterheadstartvskip
              {\vspace*{-5\baselineskip}}
% select TeX Gyre Heros for title and quote:
\usepackage{tgheros}
\renewcommand\sectfont{\sffamily\bfseries}
\renewcommand\quotefont{\sffamily\slshape}
\renewcommand\qauthorfont{\scshape}
```

```
\begin{savequote}[10pc]
 Cookies! Give me some cookies!
 \qauthor{Cookie Monster}
\end{savequote}
\chapter{A Package Test}
Adding this package changes the chapter
heading dramatically.
```

2-2-6 Adding this package changes the chapter heading dramatically.

With the quotchap package the quotation is directly integrated into the design of the chapter heading. The epigraph package by Peter Wilson has a different approach; here the quotation is typeset after the heading (using the command \epigraph or the environment epigraphs), and the heading command itself has no knowledge of it. On one hand this is more versatile; on the other it clearly means that designs that properly interact with the heading text are not possible.

The package offers a lot of configuration possibilities, typically by redefining some command or setting a dimension. A few of them are shown in the next example (but actually using the default values, so none of the redefinitions has any effect). For others you have to consult the package documentation.

```
\usepackage{epigraph}
  \setlength\epigraphwidth{.4\textwidth}
  \renewcommand\epigraphsize {\small}
  \renewcommand\epigraphflush{flushright}
  \renewcommand\sourceflush  {flushright}
```

```
\section{A Package Test}
\epigraph{Cookies! Give me some cookies!}
         {Cookie Monster}
When adding a quote, the paragraph following
it comes out indented. If you do not like
this, you have to use \verb=\noindent= at
the  beginning of this paragraph.
```

1 A Package Test

> Cookies! Give me some cookies!
>
> Cookie Monster

2-2-7 When adding a quote, the paragraph following it comes out indented. If you do not like this, you have to use \noindent at the beginning of this paragraph.

There are also mechanisms to place an epigraph onto a chapter or part heading using the command \epigraphhead; see the package documentation for details.

2.2.5 indentfirst — Indent the first paragraph after a heading

Standard LaTeX document classes and many others, following (American) English typographic tradition, suppress the indentation of the first paragraph after a display

heading. While this can be changed with an option to titlesec (see below), it can also be done through the little[1] package indentfirst by David Carlisle, regardless of whether or not the titlesec package is loaded.

2.2.6 nonumonpart — No page numbers on parts

Another often asked for adjustment is to drop page numbers on part titles. On chapter headings this can be easily manually achieved using `\thispagestyle{empty}`, but because `\parts` in many classes occupy a whole page, there is no possibility to place such a declaration.[2] To solve this without any manual work someone suggested a few lines of code, and Yvon Henel took the effort to put them into the little nonumonpart package. It works for the standard classes report and book and any other class that is derived from them. All you have to do is load the package; there are no options or other customization possibilities.

2.2.7 titlesec — A package approach to heading design

The titlesec package created by Javier Bezos provides a flexible and fairly comprehensive reimplementation of the basic heading tools offered by Standard LaTeX and is therefore a good choice if adjustments are wanted or new document classes are to be designed. It works together with most document classes in existence; notable exceptions are memoir and the KOMA-Script classes, both of which have their own tools for setting up heading structures that need to be used.

Javier's approach overcomes some of the limitations inherent in the original LaTeX tools and provides a cleaner and more generic interface. The package supports two interfaces: a simple one for smaller adjustments, which is realized mainly by options to the package, and an extended interface to make more elaborate modifications.

The basic interface

The basic interface lets you modify the font characteristics of all headings by specifying one or more options to set a font family (`rm`, `sf`, `tt`), a font series (`md`, `bf`), or a font shape (`up`, `it`, `sl`, `sc`). The title size can be influenced by selecting one of the following options: `big` (same sizes as for standard LaTeX classes), `tiny` (all headings except for chapters in text size), `medium`, or `small`, which are layouts between the two extremes. The alignment is controlled by `raggedleft`, `center`, or `raggedright`, while the vertical spacing can be reduced by specifying the option `compact` as shown later.

To modify the format of the number accompanying a heading, the command `\titlelabel` is available. Within it `\thetitle` refers to the current sectioning

[1] This package probably holds the record of "the shortest package in the LaTeX world": besides 40 lines of comments it consists of two lines of code.

[2] Well, you could try to put it into the heading title, but you will soon find that this not a good place for a number of reasons (though one can make it work with the help of the optional argument to the `\part` command).

number, such as \thesection or \thesubsection. The declaration applies to all headings, as can be seen in the next example:

1. A section

1.1. A subsection

1.1.1. A subsubsection

2-2-8 Three headings following each other, a situation you will not see very often …

```
\usepackage[sf,bf,tiny,center]{titlesec}
\titlelabel{\thetitle.\enspace}
\section{A section}
\subsection{A subsection}
\subsubsection{A subsubsection}
Three headings following each other, a situation you
will not see very often \ldots
```

\titleformat*{*cmd*}{*format*}

The basic interface offers one more command, \titleformat*, that takes two arguments. The first argument (*cmd*) is a sectioning command that we intend to modify. The second argument (*format*) contains the formatting instruction that should be applied to this particular heading. This declaration works on individual sectioning commands, and its use overwrites all font or alignment specifications given as options to the package (i.e., the options rm, it, and raggedleft in the following example). The last command used in the second argument can be a command with one argument — it receives the title text if present. In the next example we use this feature to set the \subsubsection title in small capitals (though this looks rather ugly with full-sized numbers).

1 A section

1.1 A subsection

1.1.1 A SUBSUBSECTION

2-2-9 Three headings following each other, a situation you will not see very often …

```
\usepackage[rm,it,raggedleft,tiny,compact]{titlesec}
\titleformat*{\subsubsection}{\scshape\MakeLowercase}
\section{A section}
\subsection{A subsection}
\subsubsection{A subsubsection}
Three headings following each other, a situation you
will not see very often \ldots
```

Hyphenation and line breaks in headings

In many LaTeX document classes (with or without loading titlesec), words in long headings are justified and, if necessary, hyphenated as can be seen in the next example. If this is not wanted, line breaks can be manually adjusted using \\, but then one has to repeat the heading title, without the extra formatting instruction, in the optional argument. Otherwise, the line breaks also show up in the table of contents.

1 A very long heading that shows the default behavior of LaTeX's sectioning commands

2-2-10 Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero.

```
\usepackage{lipsum,titlesec}

\section{A very long heading that
         shows the default behavior of
         \LaTeX's sectioning commands}
\lipsum[3][1-2]
```

Alternatively, one can use the option `raggedright` from the simple interface, which then applies to all heading, or use the extended interface to make a dedicated decision for each heading level separately.

1 A very long heading that shows the default behavior of LaTeX's sectioning commands

```
\usepackage[raggedright]{titlesec}
\usepackage{lipsum}

\section{A very long heading that
         shows the default behavior of
         \LaTeX's sectioning commands}
\lipsum[3][1-3]
```

2-2-11

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis.

Interpretation of heading command arguments

Two other options may offer some extra help for such cases: if you specify `newlinetospace`, then any `\\` or `\\*` in the heading text is replaced by a space before the text is passed on to the table of contents or into the running header so that it is not necessary to use the optional argument to the heading command, just because the text has explicit line breaks. The option `toctitles` changes the use of the optional argument so that it is only specifying the text for the running header while the TOC always receives the full text.

Indentation after heading

The paragraph indentation for the first paragraph following the headings can be globally specified using the package options `indentafter` or `noindentafter`. With the extended interface this can be done for individual heading levels.

Adjusting "empty" pages

If chapter headings always appear on recto pages (by internally issuing a `\cleardoublepage` command), then this often generates an empty verso page — except that this page may still contain a page number or a running header. To force such pages to be totally empty you can specify the option `clearempty`. See also the `nextpage` package discussed in Section 5.6.4 on page 418 for alternative approaches.

`\part*` in the TOC

For some reason the default for `\part*` used by titlesec is that these headings show up in the table of contents. If that is not wanted, use the option `notocpart*`. The `\part` heading is otherwise not influenced by settings for the basic interface. If you want to modify it, you must use the extended interface described below.

Fixing a TOC problem with `\part`

Another option specific to `\part` commands is `newparttoc`. This changes the entries generated in the TOC so that they can be manipulated by the titletoc package, which is normally not the case as they have a nonstandard definition in most LaTeX classes. See the discussion on page 72 for details.

The extended interface

The extended interface consists of two major commands, `\titleformat` and `\titlespacing`. They allow you to declare the "inner" format (i.e., fonts, label, alignment, ...) and the "outer" format (i.e., spacing, indentation, etc.), respectively. This scheme was adopted because people often wish to alter only one or the other aspect of the layout.

`\titleformat{`*cmd*`}[`*shape*`]{`*format*`}{`*label*`}{`*sep*`}{`*before-code*`}[`*after-code*`]`

The first argument (*cmd*) is the heading command name (e.g., `\section`) whose format is to be modified. In contrast to LaTeX's `\@startsection` (see Section 2.2.8 on page 51) this argument requires the command name — that is, with the backslash in front. The remaining arguments have the following meaning:

shape The basic shape for the heading. A number of predefined shapes are available: `hang`, the default, produces a hanging label (like `\section` in standard classes); `display` puts label and heading text on separate lines (like standard `\chapter`); while `runin` produces a run-in title (like standard `\paragraph`).
In addition, the following shapes, which have no equivalents in standard LaTeX, are provided: `frame` is similar to `display` but frames the title; `leftmargin` puts the title into the left margin, while `rightmargin` places it into the right margin. The last two shapes might conflict with `\marginpar` commands; that is, they may overlap.
A general-purpose shape is `block`, which typesets the heading as a single block. It should be preferred to `hang` for centered layouts.
Both `drop` and `wrap` wrap the first paragraph around the title, with `drop` using a fixed width for the title and `wrap` using the width of the widest title line (automatically breaking the title within the limit forced by the *left-sep* argument of `\titlespacing`).

format The declarations that are applied to the whole title — label and text. They may include only vertical material, which is typeset following the space above the heading. If you need horizontal material, it should be entered in the *label* or *before-code* argument.

label The formatting of the label, that is, the heading number. To refer to the number itself, use `\thesection` or whatever is appropriate. For defining `\chapter` headings the package offers `\chaptertitlename`, which produces `\chaptername` or `\appendixname`, depending on the position of the heading in the document.

sep Length whose value determines the distance between the label and title text. Depending on the *shape* argument, it might be a vertical or horizontal separation. For example, with the `frame` shape, it specifies the distance between the frame and heading text.

before-code Code executed immediately preceding the heading text. Its last command can take one argument, which will pick up the heading text and thus permits more complicated manipulations (see Example 2-2-15).
Since version 2.7, it is possible to load the package with the option `explicit` in which case the heading text *must* be given explicitly as `#1` inside *before-code*. This makes the declaration somewhat clearer, and you can do any manipulations directly instead of defining a command with one argument to do the job.

after-code Optional code to be executed after formatting the heading text (still within the scope of the declarations given in *format*). For `hang`, `block`, and `display`,

it is executed in vertical mode; with `runin`, it is executed in horizontal mode. For other shapes, it has no effect.

If the starred form of a heading is used, the *label* and *sep* arguments are ignored because no number is produced.

The next example shows a more old-fashioned run-in heading, for which we define only the format, not the spacing around the heading. The latter is manipulated with the `\titlespacing` command.

```
\usepackage{titlesec}
\titleformat{\section}[runin]{\normalfont\scshape}
      {\S\,\oldstylenums{\thesection}.}{.5em}{}[.]
\section{The Title}
The heading is separated from the section text by
a dot and a space of one quad.
```

2-2-12

§ 1. THE TITLE. The heading is separated from the section text by a dot and a space of one quad.

By default, LaTeX's `\section` headings are not indented (they are usually of *shape* `hang`). If you prefer a normal paragraph indentation with such a heading, you could add `\indent` before the `\S` sign or specify the indentation with the `\titlespacing` declaration, described next.

`\titlespacing*{`*cmd*`}{`*left-sep*`}{`*before-sep*`}{`*after-sep*`}[`*right-sep*`]`

The starred form of the command suppresses the paragraph indentation for the paragraph following the title, except with shapes where the heading and paragraph are combined, such as `runin` and `drop`. The *cmd* argument holds the heading command name to be manipulated. The remaining arguments are as follows:

left-sep Length specifying the increase of the left margin for headings with the `block`, `display`, `hang`, or `frame` shape. With `...margin` or `drop` shapes it specifies the width of the heading title, with `wrap` it specifies the maximum width for the title, and with `runin` it specifies the indentation before the title (negative values would make the title hang into the left margin).

before-sep Length specifying the vertical space added above the heading.

after-sep Length specifying the separation between the heading and the following paragraph. It can be a vertical or horizontal space depending on the shape deployed.

right-sep Optional length specifying an increase of the right margin, which is supported for the shapes `block`, `display`, `hang`, and `frame`.

In the case of a run-in heading, *after-sep* is the horizontal space after the heading that by default is usually noticeably wider than a normal word space. This is reasonable for headings such as the one in Example 2-2-12 but not if the heading and following text are forming a sentence in which case we want a normal word space. For this you

can use \wordsep in *after-sep*, which refers to the interword space (including stretch and shrink) of the current font.

2-2-13

… some text above.

THE MAN started to run away from the truck. He saw that he was followed by the …

```
\usepackage{titlesec}
\titleformat {\paragraph}[runin]{\normalfont\scshape}{}{0pt}{}
\titlespacing{\paragraph}{\parindent}{\medskipamount}{\wordsep}
\noindent \ldots\ some text above.
\paragraph{The man} started to run away from the truck.
He saw that he was followed by the \ldots
```

The *before-sep* and *after-sep* arguments usually receive rubber length values to allow some flexibility in the design. To simplify the declaration you can alternatively specify **f* (where *f* is a decimal factor). This is equivalent to *f* ex with some stretchability as well as a small shrinkability inside *before-sep*, and an even smaller stretchability and no shrinkability inside *after-sep*.

2-2-14

…some text before …

SECTION 1
A Title Test

Some text to prove that this paragraph is not indented and that the title has a margin of 1pc on either side.

```
\usepackage{titlesec}
\titleformat{\section}[frame]{\normalfont}
    {\footnotesize \enspace SECTION \thesection
     \enspace}{6pt}{\large\bfseries\filcenter}
\titlespacing*{\section}{1pc}{*4}{*2.3}[1pc]
\noindent \ldots some text before \ldots
\section{A Title Test}
Some text to prove that this paragraph is not indented
and that the title has a margin of 1pc on either side.
```

Spacing tools for headings

The previous example introduced \filcenter, but there are also \filleft, \filright, and \fillast — the latter produces an adjusted paragraph but centers the last line. These commands should be preferred to \raggedleft or \raggedright inside \titleformat, as the latter would cancel *left-sep* or *right-sep* set up by the \titlespacing command. Alternatively, you can use \filinner or \filouter, which resolve to \filleft or \filright, depending on the current page. However, due to TEX's asynchronous page makeup algorithm, they are supported only for headings that start a new page — for example, \chapter in most designs. See Example 2-2-17 on page 49 for a solution to this problem for other headings. Another useful spacing command we already used in Example 2-2-13 is \wordsep, which refers to the current interword space.

Spacing between consecutive headings

By default, the spacing between two consecutive headings is defined to be the *after-sep* of the first one. If this result is not desired, you can change it by specifying the option largestsep, which puts the spacing to the maximum of *after-sep* from the first heading and *before-sep* of the second.

Space reserved for chapter headings

Normally the vertical space occupied by a display heading is the sum of *before-sep*, the size of the actual heading text, and the *after-sep*; i.e., it varies depending on the number of lines in the heading. However, in some designs the text following the chapter heading should always start at the same point regardless. This can be achieved

by specifying the option `rigidchapters`. If used, *after-sep* no longer specifies the space below the heading but always measured from the top of the heading text; i.e., the sum of *before-sep* and *after-sep* defines the space reserved for the heading. Despite its name, the option applies to any heading of class `top`; see page 50.

Headings at page bottom

After a heading LaTeX tries to ensure that at least two lines from the following paragraph appear on the same page as the heading title. If this proves impossible, the heading is moved to the next page. If you think that two lines are not enough, try the option `nobottomtitles` or `nobottomtitles*`, which move headings to a new page whenever the remaining space on the page is less than the current value of `\bottomtitlespace`. (Its default is `.2\textheight`; to change its value, use `\renewcommand` rather than `\setlength`.) The starred version is preferred, as it computes the remaining space with more accuracy, unless you use headings with `drop`, `margin`, or `wrap` shapes, which may get badly placed when deploying the starred option.

Handling unusual layouts

In most heading layouts the number appears either on top or to the left of the heading text. If this placement is not appropriate, the *label* argument of `\titleformat` cannot be used. Instead, one has to exploit the fact that the *before-code* can pick up the heading text. In the next example, the command `\secformat` has one argument that defines the formatting for the heading text and number; we then call this command in the *before-code* argument of `\titleformat`. Note that the font change for the number is kept local by surrounding it with braces. Without them the changed font size might influence the title spacing in some circumstances.

```
\usepackage{titlesec}
\newcommand\secformat[1]{%
  \parbox[b]{.5\textwidth}{\filleft\bfseries #1}%
  \quad\rule[-12pt]{2pt}{70pt}\quad
  {\fontsize{60}{60}\selectfont\thesection}}
\titleformat{\section}[block]
  {\filleft\normalfont\sffamily}{}{0pt}{\secformat}
\titlespacing*{\section}{0pt}{*3}{*2}[1pc]
\section{A Title\\ on Two Lines}
In this example the heading number appears to
the right of the heading text.
```

A Title on Two Lines **1**

In this example the heading number appears to the right of the heading text.

2-2-15

The same technique can be applied to change the heading text in other ways. For example, if we want a period after the heading text, we could define

```
\newcommand\secformat[1]{#1.}
```

and then call `\secformat` in the last mandatory argument of the `\titleformat` declaration as shown in the previous example. Alternatively, we could have used the option `explicit` in which case such manipulations could have been done inline with `#1` referencing the heading text inside that argument.

Measuring the width of the title

The `wrap` shape has the capability to measure the lines in the title text and return the width of the widest line in `\titlewidth`. This capability can be extended to three

other shapes (`block`, `display`, and `hang`) by loading the package with the option `calcwidth` and then using `\titlewidth` within the arguments of `\titleformat`, as needed.

Measuring the title means trial typesetting it, and thus it is typeset twice. In some cases that can have undesirable side effects. For special requirements, the package therefore offers the command `\iftitlemeasuring`. It takes two arguments: the first is executed during the trial and the second when the heading is finally typeset.

Rules and leaders

For rules and leaders the package offers the `\titlerule` command. Used without any arguments it produces a rule of height `.4pt` spanning the full width of the column (but taking into account changes to the margins as specified with the `\titlespacing` declaration). An optional argument lets you specify a height for the produced rule. The starred form of `\titlerule` is used to produce leaders (i.e., repeated objects) instead of rules. It takes an optional *width* argument and a mandatory *text* argument. The *text* is repeatedly typeset in boxes with its natural width, unless a different *width* is specified in the optional argument. In that case, only the first and last boxes retain their natural widths to allow for proper alignment on either side.

The command `\titleline` lets you add horizontal material to arguments of `\titleformat` that expect vertical material. It takes an optional argument specifying the alignment and a mandatory argument containing the material to typeset. It produces a box of fixed width taking into account the marginal changes due to the `\titlespacing` declaration. Thus, either the material needs to contain some rubber space, or you must specify an alignment through the optional argument (allowed values are `l`, `r`, and `c`).

The `\titleline*` variant first typesets the material from its mandatory argument in a box of width `\titlewidth` (so you may have to add rubber space to this argument) and then uses this box as input to `\titleline` (i.e., aligns it according to the optional argument). Remember that you may have to use the option `calcwidth` to ensure that `\titlewidth` contains a sensible value.

In the next somewhat artificial example, which is worth studying though better not used in real life, all of these tools are applied together:

```
\usepackage[noindentafter,calcwidth]{titlesec}
\titleformat{\section}[display]
  {\filright\normalfont\bfseries\sffamily}
  {\titleline[r]{Section \Huge\thesection}}{1ex}
  {\titleline*[l]{\titlerule[1pt]}\vspace{1pt}%
   \titleline*[l]{\titlerule[2pt]}\vspace{2pt}}
  [{\titleline*[l]{\titlerule*{\tiny\LaTeX}}}]
\titlespacing{\section}{1pc}{*3}{*2}
```

Section 1

Rules and Leaders

LaTeX LaTeX LaTeX LaTeX LaTeX LaTeX LaTeX LaTeX

Note that the last `\titleline*` is surrounded by braces. Without them its optional argument would prematurely end the outer optional argument of `\titleformat`.

2-2-16

```
\section{Rules and Leaders}
Note that the last \verb=\titleline*= is
surrounded by braces. Without them its
optional argument would prematurely end the
outer optional argument of \verb=\titleformat=.
```

Breaking before a heading

Standard LaTeX considers the space before a heading to be a good place to break the page unless the heading immediately follows another heading. The penalty to break at this point is stored in the internal counter `\@secpenalty`, and in many classes it holds the value `-300` (negative values are bonus places for breaking). Because only one penalty value is available for all heading levels, there is seldom any point in modifying its setting. With titlesec, however, you can exert finer control: whenever a command `\`*name*`break` is defined (where `\`*name* is the name of a sectioning command, such as `\sectionbreak`), the latter will be used instead of adding the default penalty. For example,

```
\newcommand\sectionbreak{\clearpage}
```

would result in sections always appearing on top of a page with all pending floats being typeset first. This interface also exists for headings of class `top`. For example, you can force parts to always start on a recto page, while chapters could be set to just start a new page by using `\cleardoublepage` and `\clearpage`, respectively. However, you have to first change their class to `page` or `top`, because this is not automatically done. Heading classes are explained on page 50.

Always keeping the space above a heading

In some layouts the space above a heading must be preserved, even if the heading appears on top of a page (by default, such spaces vanish at page breaks). This can be accomplished using a definition like the following:

```
\newcommand\sectionbreak{\addpenalty{-300}\vspace*{0pt}}
```

The `\addpenalty` command indicates a (good) breakpoint, which is followed by a zero space that cannot vanish. Thus, the "before" space from the heading will appear as well at the top of the page if a break is taken at the penalty.

Special page styles

Headings that start a new page often require a special page style; e.g., `\chapter` commands in the standard styles usually use `plain` even if for other pages a different style has been set up. To accommodate adjustments titlesec offers the command `\assignpagestyle`. For example,

```
\assignpagestyle{\chapter}{empty}
```

results in pages starting a new chapter to have neither a page number nor a running header. This command works with any heading of class `top` or `page`; see page 50. There are, however, restrictions when the sectioning command was not defined with titlesec; e.g., when using the standard document classes, it works for `\chapter` but not for `\part`. For the latter you first have to redeclare a format with `\titleformat`.

Conditional heading layouts

So far we have seen how to define fixed layouts for a heading command using `\titleformat` and `\titlespacing`. The titlesec package also allows you to conditionally change the layout on verso and recto pages and to use special layouts for numberless headings (i.e., those produced by the starred form of the heading command).

This is implemented through a keyword/value syntax in the first argument of `\titleformat` and `\titlespacing`. The available keys are `name`, `page` (values

odd or even), and numberless (values true or false). In fact, the syntax we have seen so far, \titleformat{\section}{..}..., is simply an abbreviation for the general form \titleformat{name=\section}{..}....

In contrast to the spacing commands \filinner and \filouter, which can be used only with headings that start a new page, the page keyword enables you to define layouts that depend on the current page without any restriction. To specify the layout for a verso (left-hand) page, use the value even; for a recto (right-hand) page, use the value odd. Such settings only affect a document typeset in twoside mode. Otherwise, all pages are considered to be recto in LaTeX. In the following example we use a block shape and shift the heading to one side, depending on the current page. In a similar fashion you could implement headings that are placed in the margin by using the shapes leftmargin and rightmargin.

The example also shows that placing declarations into the *format* argument affects both number and title, while placing them into *before-code* affects only the title: both are in bold, but only the text is in bold italics.

2-2-17

1. ***A Head***

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna.

2. ***Another***

Lorem ipsum dolor sit amet, consectetuer

```
\usepackage{lipsum,titlesec}
\titleformat{name=\section,page=odd}[block]
   {\normalfont\bfseries}{\thesection.}{6pt}
   {\itshape\filleft}
\titleformat{name=\section,page=even}[block]
   {\normalfont\bfseries}{\thesection.}{6pt}
   {\itshape\filright}
\section{A Head}    \lipsum[1][1-4]
\section{Another}   \lipsum[1][1-4]
```

Similarly, the numberless key is used to specify that a certain \titleformat or \titlespacing declaration should apply only to headings without numbers (value true) or to those with numbers (value false). By default, a heading declaration applies to both cases, so in the example the second declaration actually overwrites part of the first declaration. To illustrate what is possible the example uses quite different designs for the two cases — do not mistake this for an attempt to show good taste. It is important to realize that neither the *label* nor the *sep* argument is ignored when numberless is set to true as seen in the example — in normal circumstances you would probably use {}{0pt} as values.

2-2-18

1. A Head

Some text to fill the page. Some text to fill the page.

— *Another*

Some text to fill this line.

```
\usepackage{titlesec}
\titleformat{name=\section}[block]
  {\normalfont\bfseries}{\thesection.}{6pt}{\filright}
\titleformat{name=\section,numberless=true}[block]
  {\normalfont}{---}{12pt}{\itshape\filcenter}
\section{A Head}
Some text to fill the page. Some text to fill the page.
\section*{Another}
Some text to fill this line.
```

Changing the heading hierarchy

The commands described so far are intended to adjust the formatting and spacing of existing heading commands. With the `\titleclass` declaration it is possible to define new headings.

`\titleclass{`*cmd*`}{`*class*`}`
`\titleclass{`*cmd*`}{`*class*`}[`*parent-level-cmd*`]`
`\titleclass{`*cmd*`}[`*start-level*`]{`*class*`}` (with `loadonly` option)

There are three classes of headings: the `page` class contains headings that fill a full page (like `\part` in LaTeX's `report` and `book` document classes); the `top` class contains headings that start a new page and thus appear at the top of a page; and all other headings are considered to be part of the `straight` class.

Used without any optional argument, the `\titleclass` declaration simply changes the heading class of an existing heading *cmd*. For example,

```
\titleclass\section{top}
```

would result in sections always starting a new page. Note, however, that the existing *cmd* should have been defined using `titlesec` or at least should have been given a format with `\titleformat` in order to work. Otherwise you get an error message.

If this declaration is used with the optional *parent-level-cmd* argument, you introduce a new heading level below *parent-level-cmd*. Any existing heading command at this level is moved one level down in the hierarchy. For example,

```
\titleclass\subchapter{straight}[\chapter]
```

introduces the heading `\subchapter` between `\chapter` and `\section`. The declaration does not define any layout for this heading (which needs to be defined by an additional `\titleformat` and `\titlespacing` command), nor does it initialize the necessary counter. Most likely you also want to update the counter representation for `\section`:

```
\titleformat{\subchapter}{..}...    \titlespacing{\subchapter}{..}...
\newcounter{subchapter}
\renewcommand\thesubchapter{\thechapter.\arabic{subchapter}}
\renewcommand\thesection{\thesubchapter.\arabic{section}}
```

The third variant of `\titleclass` is needed only when you want to build a heading structure from scratch — for example, when you are designing a completely new document class that is not based on one of the standard classes. In that case load the package with the option `loadonly` so that the package will make no attempt to interpret existing heading commands so as to extract their current layout. You can then start building heading commands, as in the following example:

```
\titleclass\Ahead[0]{top}
\titleclass\Bhead{straight}[\Ahead]
\titleclass\Chead{straight}[\Bhead]
```

```
\newcounter{Ahead} \newcounter{Bhead} \newcounter{Chead}
\renewcommand\theBhead{\theAhead-\arabic{Bhead}}
\renewcommand\theChead{\theBhead-\arabic{Chead}}
\titleformat{name=\Ahead}{..}...  \titlespacing{name=\Ahead}{..}...
\titleformat{name=\Bhead}{..}...      ...
```

The *start-level* is usually `0` or `-1`; see the introduction in Section 2.2 for its meaning. There should be precisely one `\titleclass` declaration that uses this particular optional argument.

If you intend to build your own document classes in this way, take a look at the documentation accompanying the titlesec package. It contains additional examples and offers further tips and tricks.

2.2.8 Formatting headings — LaTeX's internal low-level methods

While it is recommended to use the higher-level interfaces provided by titlesec or those defined by KOMA-Script or the memoir class, it is useful to have a basic understanding of the interfaces defined in the LaTeX kernel, given that these interfaces are still in use in many document classes.[1]

LaTeX provides a generic command called `\@startsection` that can be used to define a wide variety of heading layouts. If the desired layout is not achievable that way, then `\secdef` can be used to produce sectioning formats with arbitrary layout. It is used by the standard classes to define `\chapter` and `\part` headings.

The generic command `\@startsection` allows both types of headings to be defined. Its syntax and argument description are as follows:

`\@startsection{`*name*`}{`*level*`}{`*indent*`}{`*beforeskip*`}{`*afterskip*`}{`*style*`}`

name The name used to refer to the heading counter[2] for numbered headings and to define the command that generates a running header or footer (see page 390). For example, *name* would be the counter name, `\the`*name* would be the command to display the current heading number, and `\`*name*`mark` would be the command for running headers. In most circumstances the *name* will be identical to the name of the sectioning command being defined, without the preceding backslash — but this is no requirement.

level A number denoting the depth level of the sectioning command. This level is used to decide whether the sectioning command gets a number (if the level is less than or equal to `secnumdepth`; see Section 2.2.1 on page 34) or shows up in the table of contents (if the value is less or equal to `tocdepth`; see Section 2.3.4 on page 71). It should therefore reflect the position in the command hierarchy of sectioning commands, where the outermost sectioning command has level zero.[3]

indent The indentation of the heading with respect to the left margin. By making the value negative, the heading starts in the outer margin. Making it positive indents all lines of the heading by this amount.

beforeskip The absolute value of this parameter defines the space to be left in front of the heading. If the parameter is negative, then the indentation of the paragraph following the heading is suppressed. This dimension is a rubber length; that is, it can take a stretch and shrink component. Note that LaTeX starts a new paragraph before the heading so that additionally the value of `\parskip` is added to the space in front.

[1] The whole section is set in a smaller font to indicate that is more a reference — helpful mainly when studying existing code.

[2] This counter must exist; it is not defined automatically.

[3] In the book and report classes, the `\part` command actually has level −1 (see Table 2.1).

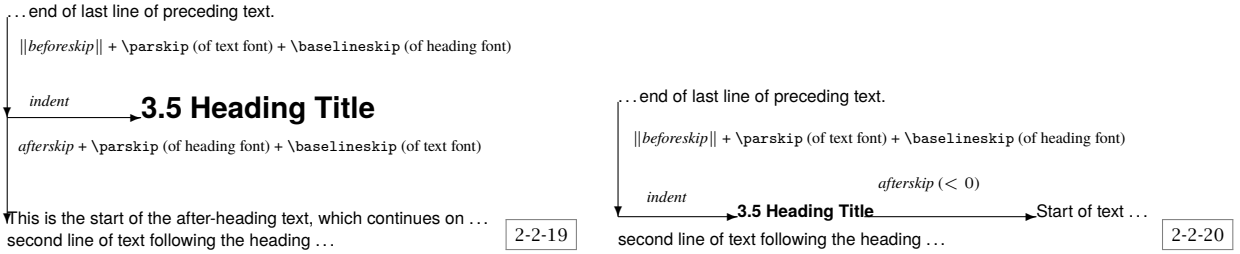


Figure 2.1: The layout for display and run-in headings (produced by layouts)

afterskip The space to be left following a heading. It is the vertical space after a display heading or the horizontal space after a run-in heading. The sign of *afterskip* controls whether a display heading (*afterskip* $>$ 0) or a run-in heading (*afterskip* $\leq$ 0) is produced. In the first case a new paragraph is started so that the value of `\parskip` is added to the space after the heading. An unpleasant side effect of this parameter coupling is that it is impossible to define a display heading with an effective "after space" of less than `\parskip` using the `\@startsection` command. When you try to compensate for a positive `\parskip` value by using a negative *afterskip*, you change the display heading into a run-in heading.

style The style of the heading text. This argument can take any instruction that influences the typesetting of text, such as `\raggedright`, `\Large`, or `\bfseries` (see the examples below).

Figure 2.1 shows these parameters graphically for the case of display and run-in headings, respectively. As an example we redefine `\subsection` to be set in normal-sized italic with the separation from the preceding text being exactly one baseline. The separation from the text following is one-half baseline, and this text is not indented.

```
\makeatletter
\renewcommand\subsection{\@startsection
  {subsection}{2}{0mm}%                % name, level, indent
  {-\baselineskip}{0.5\baselineskip}% % beforeskip, afterskip
  {\normalfont\normalsize\itshape}}%  % style
\makeatother
\ldots\ some text above.
\subsection{Subsection Heading}
The first paragraph following the redefined subsection
heading \ldots   \par   And a second one (indented).
```

2-2-21

. . . some text above.

4.1 Subsection Heading

The first paragraph following the redefined subsection heading . . .

And a second one (indented).

The first argument to `\@startsection` is the string `subsection` to denote that we use the corresponding counter for heading numbers. In the sectional hierarchy we are at level two. The third argument is `0mm` because the heading should start at the left margin.

The absolute value of the fourth argument (*beforeskip*) specifies that a distance equal to one baseline must be left in front of the heading and, because the parameter is negative, that the indentation of the paragraph following the heading should be suppressed.

The absolute value of the fifth parameter (*afterskip*) specifies that a distance equal to one-half baseline must be left following the heading and, because the parameter is positive, that a display heading has to be produced. Finally, according to the sixth parameter, the heading should be typeset in an italic font using a size equal to the normal document type size.

In fact, the redefinition is a bit too simplistic because, as mentioned earlier, on top of the absolute value of *beforeskip* and *afterskip*, LaTeX always adds the current value of `\parskip`. Thus, in layouts where this parameter is nonzero, we need to subtract it to achieve the desired separation.

Other simple heading style changes

Which commands can be used for setting the styles of the heading texts in the *style* argument of the `\@startsection` command? Apart from the font-changing directives (see Chapter 9), few instructions can be used here. A `\centering` command produces a centered display heading, and a

\raggedright declaration makes the text left justified. The use of \raggedleft is possible, but may give somewhat strange results. You can also use \hrule, \medskip, \newpage, or similar commands that introduce local changes.

Complex heading layout definitions

In the standard LaTeX classes the highest-level sectioning commands \part and \chapter produce their titles without using \@startsection because their layout cannot be produced with that command. Similarly, you may also want to construct sectioning commands without limitations. In this case you must follow a few conventions to allow LaTeX to take all the necessary typesetting actions when executing them.

The command \secdef can help you when defining such commands by providing an easy interface to the three possible forms of section headings. With the definition

```
\newcommand\myhead{\secdef\myheadA\myheadB}
```

the following actions take place:

\myhead{*title*}	invokes	\myheadA[*title*]{*title*}
\myhead[*toc-entry*]{*title*}	invokes	\myheadA[*toc-entry*]{*title*}
\myhead*{*title*}	invokes	\myheadB{*title*}

The commands you have to provide are a (re)definition[1] of \myhead and a definition of the commands named \myheadA or \myheadB, respectively. Note that \myheadA has an optional argument containing the text to be entered in the table of contents .toc file, while the second (mandatory) argument, as well as the single argument to \myheadB, specifies the heading text to be typeset. Thus, the definitions must have the following structure:

```
\newcommand\myhead{ ... \secdef \myheadA \myheadB }
\newcommand\myheadA[2][default]{ ... }
\newcommand\myheadB[1]{ ... }
```

An explicit example is a simplified variant of \appendix. It redefines the \section command to produce headings for appendices (by invoking either the command \Appendix or \sAppendix), changing the presentation of the section counter and resetting it to zero. The modified \section command also starts a new page (with all deferred floats placed), which is typeset with a special page style (see Chapter 5) and with top floats suppressed. The indentation of the first paragraph in a section is also suppressed by using the low-level kernel command \@afterheading and setting the Boolean switch @afterindent to false. For details on the use of these commands, see the \chapter implementation in the standard classes (file classes.dtx).

```
\makeatletter
\renewcommand\appendix{%
   \renewcommand\section{%                    % Redefinition of \section...
      \clearpage\thispagestyle{plain}%        %  new page, folio bottom
      \suppressfloats[t]\@afterindentfalse    %  no top floats, no indent
      \secdef\Appendix\sAppendix}%            %  call \Appendix or \sAppendix
   \setcounter{section}{0}\renewcommand\thesection{\Alph{section}}}
```

In the definition below you can see how \Appendix advances the section counter using the \refstepcounter command (the latter also resets all subsidiary counters and defines the "current reference string"; see Section 2.4). It writes a line into the .toc file with the \addcontentsline command, formats the heading title, and saves the title for running heads and/or feet by calling \sectionmark. The \@afterheading command in the later part of the definition handles the indentation of the paragraph following the heading.

```
\newcommand\Appendix[2][?]{%                 % Complex form:
   \refstepcounter{section}%                 % step counter/ set label
   \addcontentsline{toc}{appendix}%          % generate toc entry
      {\protect\numberline{\appendixname~\thesection}#1}%
   {\raggedleft\large\bfseries \appendixname\ % typeset the title
    \thesection\par \centering#2\par}%       %   and number
```

[1] Redefinition in case you change an existing heading command such as \part in the preamble of your document.

```
    \sectionmark{#1}%                          % add to running header
    \@afterheading                             % prepare indentation handling
    \addvspace{\baselineskip}}                 % space after heading
```

The `\sAppendix` command (starred form) performs only the formatting.

```
\newcommand\sAppendix[1]{%                     % Simplified (starred) form
   {\raggedleft\large\bfseries\appendixname\par \centering#1\par}%
    \@afterheading\addvspace{\baselineskip}}
\makeatother
```

Applying these definitions produces the following output:

Appendix A
The list of all commands

Then follows the text of the first section in the appendix. Some more text in the appendix.

```
% Example needs commands introduced above!
\appendix
\section{The list of all commands}

Then follows the text of the first section in
the appendix. Some more text in the appendix.
```

2-2-22

Do not forget that the example shown above represents only a simplified version of a redefined `\section` command. Among other things, we did not take into account the `secnumdepth` counter, which contains the numbering threshold. You might also have to foresee code dealing with various types of document formats, such as one- and two-column output or one- and two-sided printing. Also missing is an appropriate definition for `\l@appendix`, which is called in the table of contents because of the `\addcontentsline`. This is discussed at the beginning of Section 2.3.4 on page 70.

2.3 Table of contents structures

A *table of contents* (TOC) is a special list in which the titles of the section units are listed, usually together with the page numbers indicating the start of the sections. This list can be rather complicated if units from several nesting levels are included, and it should be formatted carefully because it plays an important rôle as a navigation aid for the reader.

Similar lists exist containing reference information about the floating elements in a document — namely, the *list of tables* and the *list of figures*. The structure of these lists is usually simpler, as their contents, the captions of the floating elements, are normally all on the same level (but see subfloats in Section 7.5).

Standard LaTeX can automatically create these three contents lists. By default, LaTeX enters text from one of the arguments of each sectioning command into the `.toc` file. While information from all sectioning levels is added to the `.toc` file, not all of them are used when producing the table of contents. The level down to which the heading information is displayed is controlled by the counter `tocdepth`. It can be changed, for example, with the following declaration:

```
\setcounter{tocdepth}{1}
```

In this case section heading information down to the first level (e.g., in the report class part, chapter, and section) will be shown.

Granular control is possible

This counter globally defines which entries are typeset in the table of contents. Sometimes, however, more granular control is necessary; e.g., you may want to show

less or more heading levels in an appendix, etc. For such use cases, you may want to try the package tocvsec2 by Peter Wilson. It provides commands to adjust the level within the document.

Similarly, LaTeX maintains two more files, one for the list of figures (`.lof`) and one for the list of tables (`.lot`), which contain the text specified as the argument of the `\caption` command for figures and tables.

When using the `\tableofcontents`, `\listoffigures`, or `\listoftables`, the information written into these files during a previous LaTeX run is read and typeset (normally at the beginning of a document), and at the end of the run newly collected information is written back to the files.

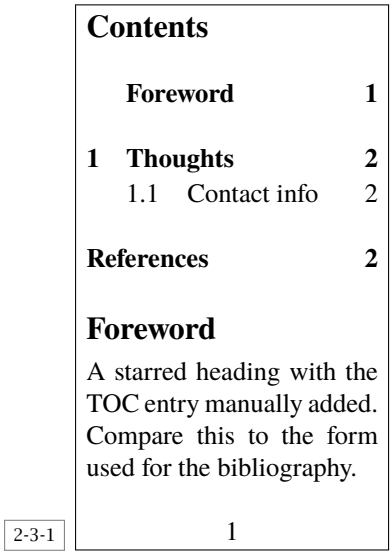
A TOC needs two, sometimes even three, LaTeX runs

To generate these cross-reference tables, it is therefore always necessary to run LaTeX at least twice — once to collect the relevant information, and a second time to read back the information and typeset it in the correct place in the document. Because of the additional material to be typeset in the second run, the cross-referencing information may change, making a third LaTeX run necessary. This is one of the reasons for the tradition of using different page-numbering systems for the front matter and the main text: in the days of hand typesetting any additional iteration made the final product much more expensive.

Normally the contents files are generated automatically by LaTeX by internally using the commands `\addcontentsline`, `\addtocontents`, and `\numberline`; see Section 2.3.4 on page 70. With some care this interface can also be used to enter information directly into these files to complement the actions of standard LaTeX.

Adding arbitrary starred headings to the TOC

For instance, in the case of the starred form of the section commands, no information is written to the `.toc` file. If you do not want a heading number (starred form) but you do want an entry in the `.toc` file, you can use `\addcontentsline` with or without `\numberline` as shown in the following example.



2-3-1

```
\tableofcontents
\section*{Foreword}
\addcontentsline{toc}{section}
   {\protect\numberline{}Foreword}
A starred heading with the TOC entry
manually added. Compare this to the
form used for the  bibliography.

\section{Thoughts}
We find all in \cite{k1}.
\subsection{Contact info}
E-mail Ben at \cite{k2}.
\begin{thebibliography}{9}
\addcontentsline{toc}{section}
   {\refname}
\bibitem{k1} Ben User, Some day will
             never come, 2010
\bibitem{k2} BUser@earth.info
\end{thebibliography}
```

Using \numberline as in the "Foreword" produces an indented "section" entry in the table of contents, leaving the space where the section number would go free. The \protect in front is required in this case; see page 70 for more details. Omitting the \numberline command (as was done for the bibliography entry) would typeset the heading flush left instead. Adding a similar line after the start of the theindex means that the "Index" will be listed in the table of contents. Unfortunately, this approach cannot be used to get the list of figures or tables into the table of contents because \listoffigures or \listoftables might generate a listing of several pages, and consequently the page number picked up by \addcontentsline might be wrong. And putting it before the command does not help either, because often these list commands start a new page. One potential solution is to copy the command definition from the class file and put \addcontentsline directly into it.

Bibliography or index in tables of contents

In the case of standard classes or close derivatives, you can use the tocbibind package created by Peter Wilson to get the "List of...", "Index", or "Bibliography" section listed in the table of contents without further additions to the source. The package offers a number of options such as notbib, notindex, nottoc, notlof, and notlot (do not add the corresponding entry to the table of contents).

Numbered headings for bibliography or index

There also exist the options numbib and numindex (number the corresponding heading), and with section you can ask for section instead of chapter headings in document classes like report or book.

An oddity better turned off

By default the "Contents" section is listed within the table of contents, which is seldom desirable — use the option nottoc to disable this behavior.

* * * * *

There are a number of packages that extend or alter standard LaTeX's table of contents mechanism. The hyperref package changes the internals to support hyperlink anchors; in particular, this changes the internal contents file structures. It is briefly touched upon on page 72; an extensive coverage of that package is found in Section 2.4.6 on page 96.

The tocdata package provides an interface for adding special data such as author names to the contents files. It is discussed in the next section. We will then turn to customizing the design of such lists with the help of the titletoc package. There are alternative packages for this available, e.g., tocloft by Peter Wilson or tocstyle by Markus Kohm, but titletoc provides a good general-purpose interface suitable for most needs, so we concentrate on that.

The final section concerned with contents file data discusses the low-level interface already provided by LaTeX and is included mainly for reference (in a smaller font) because one often find its commands in older class files.

2.3.1 tocdata — Providing extra data for the TOC

In anthologies or other multi-author works it is quite common to list the different authors in the table of contents next to their entries. The package tocdata by Brian Dunn provides a framework for this that enables you to place such data into the

typeset TOC entry just before the page number. The package works well with most document classes and supports TOC packages such as titletoc or tocloft.

In the next example we have added author names to the two subsections; the section itself shows no extra data. The extra data is formatted with the help of the command `\tocdataformat`, which by default sets the material in a small italic font. Here we added color and an em-dash.

Contents

1 On Cookies

1.1 Preparing cookies

Text of his recipes …

1.2 Eating cookies

2-3-2 How to do it …

```
\usepackage{color,tocdata}
\renewcommand\tocdataformat[1]{\textnormal{%
    \textcolor{blue}{--- \small\itshape#1}}}
\tableofcontents

\section{On Cookies}
\tocdata{toc}{Ben User}
\subsection{Preparing cookies}
Text of his recipes \ldots

\tocdata{toc}{Cookie Monster}
\subsection{Eating cookies}
How to do it \ldots
```

In a similar fashion you can add to the list of figures or tables to indicate the artist who made a certain picture or the source of the table data, etc. All you need to do is to specify in the first argument to `\tocdata` the correct target destination file extension, e.g., `lof` for the list of figures or `lot` for the list of tables.

The `\tocdata` command used in the previous example enables you to add data to any "TOC-like" file, but often you also want to provide this information within your document as well.

For such use cases the package offers a set of special commands that combine `\tocdata` with a heading or a caption command. We show the syntax for the `\part` heading, but corresponding commands exist for `\chapter` (if supported by the document class), `\section`, and `\subsection` headings.

```
\partauthor[list-entry]{title}[prefix]{first}{last}[suffix]
\partauthor*           {title}[prefix]{first}{last}[suffix]
```

The first form executes the following set of commands for you

```
\todata{toc}{first  last}
\part[list-entry]{title\nopagebreak
                  \tocdatapartprint{prefix}{first}{last}{suffix}}
\index{last, first}
```

while the star form on the second line omits the `\tocdata`, since the heading is not written to the table of contents. The `\tocdatapartprint` command formats the name and adds it as part of the heading title. By redefining this command, various

layouts can be realized. Note that *prefix* and *suffix* are used only there — the `\tocdata` and `\index` commands receive only *first* and *last*.

While *first* is a mandatory argument, it can be left empty if the author has no first name. In this case, the comma in the `\index` is automatically dropped too as shown in the example.

Contents

1 On Cookies

1.1 Preparing cookies

— Sir Ben User

Text of his recipes …

1.2 Eating cookies

— Cookie Monster!!

How to do it …

Index


```
\usepackage{makeidx}
  \makeindex % enable indexing
% save some space in the index:
  \renewcommand\indexspace{\par\vspace{2pt}}

\usepackage{tocdata}

\tableofcontents \smallskip

\section{On Cookies}
\subsectionauthor{Preparing cookies}
                 [Sir]{Ben}{User}
Text of his recipes \ldots

\subsectionauthor{Eating cookies}
                 {}{Cookie Monster}[!!]
How to do it \ldots

\printindex
```

2-3-3

For captions of figures (or tables) two commands exist with a syntax similar to `\partauthor`, but with one further optional *extra-text* argument. They are intended to be used instead of the normal `\caption` command:

`\captionartist[`*list-entry*`]{`*title*`}[`*extra-text*`][`*prefix*`]{`*first*`}{`*last*`}[`*suffix*`]`

The arguments *list-entry* and *title* correspond to the usual `\caption` arguments, and *first* and *last* are used to add the artist name to the list of figures and produce an index entry (if an index is made). Again, *prefix* and *suffix* are used only when displaying the artist name as part of the float. Finally the *extra-text* allows you to place additional information next to the caption title that does not show up in the list of figures.

Note that if you want to use the optional *prefix* but not the *extra-text*, you need to supply an empty optional argument for the latter to identify for LaTeX which is which.

To influence justification of the name there are a number of declarations available of the form `\tdartist...` where `...` is either `justify`, `left`, `center`, or `right`, and for the additional text you have `\tdartisttext...` with the same possibilities.

To change the formatting in more drastic ways, you can alternatively redefine the commands `\tocdataartistprint` (receiving *prefix*, *first*, *last*, and *suffix* as arguments to format the name) and `\tocdataartisttextprint` (responsible for

formatting *extra-text*). See the package documentation for details.

Sebastian Rahtz (1955–2016)

This has been already used
in the first edition of TLC

Figure 1: A cat

2-3-4

```
\usepackage{graphicx,tocdata}   \tdartistright
\begin{figure}
  \centering
  \includegraphics{cat}
  \captionartist{A cat}[This has been already used\\
                        in the first edition of TLC]
                       {Sebastian}{Rahtz}[ (1955--2016)]
\end{figure}
```

Instead of the command `\captionartist` you can use `\captionauthor` with exactly the same syntax (and corresponding configuration commands). The difference between the two is the default formatting: `\captionartist` typesets the name centered, whereas `\captionauthor` places it flush right. The latter may look nicer for wide pictures.

If you use the caption package, which supports the `\caption*` command, then `\captionartist` and `\captionauthor` will also accept a star.

2.3.2 titletoc — A high-level approach to contents list design

The titletoc package written by Javier Bezos was originally developed as a companion package to titlesec but can be used on its own. It implements its own interface to lay out contents structures, thereby avoiding some of the limitations of the original LaTeX code for this task. This makes it a good candidate when adjustments of such lists are necessary when a new class is being developed.

Relation to standard LaTeX

The actual generation of external contents files and their syntax is left unchanged so that it works nicely with other packages generating such files. There is one exception, however: contents files should end with the command `\contentsfinish`. For the standard file extensions `.toc`, `.lof`, and `.lot`, this is handled automatically. But if you provide your own type of contents lists (see Section 2.3.4), you have to announce it to titletoc, as in the following example:

```
\contentsuse{example}{xmp}
```

Designing the layout for a single contents list entry

A single contents list entry normally consists of one or more lines of text, typically starting with a label (e.g., the heading number) followed by the heading title and finishing off with a page number. Typically, the page number is pushed to the right edge so that page numbers from different entries align. Thus, there is normally a gap between title and page number, which is filled either by white space or by some leaders, e.g., some dots or a line.

Standard LaTeX already supports that type of design with some flexibility in allowing for indentation at the left and right of all lines. In addition, the start of the

first line as well as the endpoint of the last line can be moved (typically to place both the label and the page number outside of the title text block).

A typical multiline entry could look like this:

3.11 This is a sample section entry which has been deliberately made very long so that it spans three lines to exhibit the handling of the first and the last line in the entry 27

As you see, the entry is indented on both sides with the entry label placed into the available white space. The heading title is set ragged right in sans serif, and the page number is separated from the text block using a row of leader dots and again placed outside of the block.

Standard (dotted) layouts

The titletoc package supports this type of standard layout, but compared to standard LaTeX offers more convenient ways to customize it. In addition, it supports other layouts such as running lower-level heading entries together in a single paragraph and, as a nice add-on, supports partial table of contents lists so that you can provide chapter tables, etc. For the most common case, i.e., the layout shown above, it offers the `\dottedcontents` declaration.

`\dottedcontents{`*type*`}[`*left-indent*`]{`*before-code*`}{`*label-width*`}{`*leader-width*`}`

The first argument of `\dottedcontents` contains the *type* of contents entry for which we set up the layout — normally the name of the heading command without a backslash or the name of the float environment, e.g., `figure`. In other words, for each *type* of sectioning command that can appear in the document, we need one `\dottedcontents` (or alternatively `\titlecontents` discussed below) declaration.[1] The remaining arguments have the following meaning:

left-indent The indentation from the left margin for all lines of the entry. It should normally be wider than the *label-width* argument because the label is placed into that space. Even though this argument has to be given in square brackets, it is *not* optional in the current package release (and probably never will become one)!

before-code Code to be executed before the entry is typeset. It can be used to provide vertical space, such as by using `\addvspace`, and to set up formatting directives, such as font changes, for the whole entry. You can also use `\fillleft`, `\filright`, `\filcenter`, or `\fillast`, already known from the titlesec package, at this point.

label-width Nominal width of the label, i.e., the label starts to left of the first line offset by this amount. Thus, the value should be wide enough to comfortably

[1] The package honors existing *type* declarations made, for example, by the document class even if they are defined using the standard LaTeX interface. Thus, it can be used to change the layout of only some types.

hold the label material for this type. Problematic cases with varying label widths and possible solutions are discussed on pages 63 (`\contentspush`) and 73.

leaders-width Distance between two dots in the leaders on the last line of the entry.

For example, the entry above was typeset using the following declaration:

```
\dottedcontents{section}[40pt]{\normalfont\sffamily\filright}{24pt}{6pt}
```

i.e., we have an indentation of `40pt` from the left margin with the label starting `16pt` from the margin[1] and occupying `24pt`. The whole entry is set in sans serif and ragged right (via `\filright`), and each leader dot occupies `6pt` of space.

You may wonder where the indentation on the right (for all lines but the last) comes from and why it is not available as an argument to `\dottedcontents`. The main reason is that in nearly all designs its value is the same for all entry types, and thus providing it as an argument on the entry level would cumbersome and error prone. In most document classes the default is wide enough to contain up to three digits in the document body font. If that is not enough (or too much), it can be globally (or locally) changed with a `\contentsmargin` declaration.

`\contentsmargin[`*correction*`]{`*right-sep*`}`

This declaration shortens all entry lines by *right-sep*. On the last line the page number is typeset in that space, so if it is too small, the entry and page number may overlap. In addition, the optional *correction* argument is added to all lines of an entry except the last. This argument can, for example, be used to fine-tune the contents layout so that dots from a row of leaders align with the text of previous lines in a multiline entry if the entry is set justified.

In the unlikely case that there is a need to have different *right-sep* values for different entry types, then the solution is to place this command inside the *before-code* of `\dottedcontents` or `\titlecontents`. It is then local to that entry type.

More complicated layouts

While `\dottedcontents` works well in many cases, it clearly has its limitations and cannot be used if you do not want any leaders or other typographic adjustments that go beyond setting the font or the indentation. For such cases titletoc offers the `\titlecontents` declaration and a few helper commands to be used within its arguments.

`\titlecontents{`*type*`}[`*left-indent*`]{`*before-code*`}{`*numbered-entry-format*`}`
`{`*numberless-entry-format*`}{`*page-format*`}[`*below-code*`]`

The first three arguments *type*, *left-indent*, and *before-code* are the same as the corresponding ones for `\dottedcontents` and are described there. However, the remaining ones differ. Instead of simply specifying the width for the label we have

[1] In other words, *left-indent* minus *label-width*, i.e., 40pt – 24pt in this case.

now two arguments that allow us to explicitly define how the label and the title text should be formatted and what should happen if the label is empty. This means you have way more design possibilities at the cost of specifying more code.

numbered-entry-format Code to format the entry including its number. It is executed in horizontal mode (after setting up the indentation). The last token can be a command with one argument, in which case it receives the entry *text* as its argument. The unformatted heading number is available in the `\thecontentslabel` command, but see below for other possibilities to access and place it.

numberless-entry-format Code to format the entry if the current entry does not contain a number. Again, the last token may be a command with one argument.

Instead of specifying the *leader-width*, we now have an argument in which we have to define exactly what should happen after the title text and how the page number should be formatted. Finally, there is a further optional argument to be executed after the entry is typeset.

page-format Code that is executed after formatting the entry but while still being in horizontal mode. It is normally used to add some filling material, such as a dotted line, and to attach the page number stored in `\thecontentspage`. You can use the `\titlerule` command, discussed on page 47, to produce leaders.

below-code Optional code to be executed in vertical mode after the entry is typeset — for example, to add some extra vertical space after the entry.

To help with placing and formatting the heading and page numbers, the titletoc package offers two useful tools: `\contentslabel` and `\contentspage`.

`\contentslabel[`*text*`]{`*size*`}` `\contentspage[`*text*`]`

The purpose of the `\contentslabel` command is to typeset the *text* (which by default contains `\thecontentslabel`) left aligned in a box of width *size* and to place that box to the left of the current position. Thus, if you use this command in the *numbered-entry-format* argument of `\titlecontents`, then the number is placed in front of the entry text into the margin or indentation set up by *left-indent*. For a more refined layout you can use the optional argument to specify your own formatting usually involving `\thecontentslabel`.

In a similar fashion `\contentspage` typesets *text* (which by default contains `\thecontentspage`) right aligned in a box and arranges for the box to be placed to the right of the current position but without taking up space. Thus, if placed at the right end of a line, the box extends into the margin. In this case, however, no mandatory argument specifies the box size: it is the same for all entries. Its value is the same as the space found to the right of all entries and can be set by the command `\contentsmargin` described below.

Package options for `\contentslabel`

The package offers three options to influence the default outcome of the `\contentslabel` command when used without the *text* argument. With the option `rightlabels` the heading number is right aligned in the space, while `leftlabels`

(the default) makes it left aligned. You can also specify `dotinlabels` to always add a period after the number.

Instead of indenting the whole entry and then moving some material into the left margin using `\contentslabel`, you can make use of `\contentspush` to achieve a similar effect.

`\contentspush{`*text*`}`

This command typesets *text* and then increases the *left-indent* by the width of *text* for all additional lines of the entry (if any). As a consequence, the indentation will vary if the width of the *text* changes. In many cases such variation is not desirable, but in some cases other solutions give even worse results. Consider the case of a document with many chapters, each containing dozens of sections. A rigid *left-indent* needs to be able to hold the widest number, which may have five or six digits. In that case a label like "1.1" comes out unduly separated from its entry text. Given below is a solution that grows with the size of the entry number:

```
\usepackage{titletoc}
\titlecontents{section}[0pt]{\addvspace{2pt}\filright}
              {\contentspush{\thecontentslabel\enspace }}
              {}{~\hrulefill\contentspage}
\contentsline{section}{\numberline{12.8}Some section that
                       is wrapped in the TOC}{87}{}%
\contentsline{section}{\numberline{12.9}Another section}{88}{}%
\contentsline{section}{\numberline{12.10}And yet another
                       wrapping section}{90}{}%
\contentsline{section}{\numberline{12.11}Final section}{92}{}%
\contentsfinish
```

12.8 Some section that is wrapped in the TOC — 87
12.9 Another section — 88
12.10 And yet another wrapping section — 90
12.11 Final section — 92

2-3-5

A few design examples

A note on the examples in this and the next section

For the examples in this section we copied some parts of the original `.toc` file generated by LaTeX for this book (Chapter 2 and parts of Chapter 3) into a file we called `partial.toc` and manually added a `\contentsfinish` command at the end. Inside the examples we can then load this file with `\input`. Of course, in a real document you would use the command `\tableofcontents` instead so that the `.toc` file for *your* document is loaded and processed.

In our first example we provide a new formatting for chapter entries, while keeping the formatting for the section entries as defined by the standard LaTeX document class. The chapter entries are now set ragged right (`\filright`) in bold typeface, get one pica space above, followed by a thick rule. The actual entry is indented by six picas. In that space we typeset the word "Chapter" in small caps followed by a space and the chapter number (`\thecontentslabel`) using the `\contentslabel` directive with its optional argument. There is no special handling for entries without numbers, so they would be formatted with an indentation of six picas. We fill the remaining space using `\hfill` and typeset the page number in the margin via `\contentspage`.

Finally, after the entry we add another two points of space so that the entry is slightly separated from any section entry following.


```
\usepackage[dotinlabels]{titletoc}
\titlecontents{chapter} [5pc]
   {\addvspace{1pc}\bfseries
    \titlerule[2pt]\filright}
   {\contentslabel
      [\textsc{\chaptername}\
       \thecontentslabel]{5pc}}
   {}{\hfill\contentspage}
   [\addvspace{2pt}]
% Show only chapter/section entries:
\setcounter{tocdepth}{1}
\input{partial.toc}
```

2-3-6

In our second example we typeset the chapter title in sans serif with the chapter and page numbers on the left and right. Any free space is filled with a rule on the baseline, and we provide a bit of extra space above and below the chapter line. The section headings are shown slightly indented; for them the page numbers are suppressed. All numbers are formatted using oldstyle numerals.


```
\usepackage{titletoc}
\titlecontents{chapter}[0pc]
  {\addvspace{6pt}}
  {\large\sffamily
   \oldstylenums{\thecontentslabel}
   \ \hrulefill\ }{}
  {\large\sffamily\ \hrulefill\
   \oldstylenums{\thecontentspage}}
  [\addvspace{2pt}]
\titlecontents{section} [1pc]{}
  {\oldstylenums{\thecontentslabel}
    -- }{}{}
\setcounter{tocdepth}{1}
\input{partial.toc}
```

2-3-7

The third example and final example for now puts the page numbers in focus; they are printed on the left, while the normal heading numbers are suppressed. The chapter title is placed on the right by filling the available space with `\dotfill`. Section titles are left aligned and separated with an en-dash from the page number. Note that we use `\enspace` instead of a normal space around it so that this space does not stretch or shrink if the section title is longer than a single line.

2-3-8

```
\usepackage{titletoc}
\titlecontents{chapter}[2pc]
  {\addvspace{5pt}}
  {\large\bfseries
   \contentslabel[\hfill
     \thecontentspage]{2pc}\dotfill
   }{}{}
  [\addvspace{2pt}]
\titlecontents{section}[2pc]{}
  {\contentslabel[\hfill
        \thecontentspage]{2pc}%
   \enspace --\enspace }{}{}
\setcounter{tocdepth}{1}
\input{partial.toc}
```

Note that none of the previous examples have provisions to format headings that are unnumbered; i.e., the third mandatory argument of the `\titlecontents` command was always left empty. This was done because the sample data contains only numbered headings and it saved space to not provide formatting instructions for unnumbered headings that are never used. However, in real life you better think about how such entries should be displayed as well.

Contents entries combined in a paragraph

Standard LaTeX only supports contents entries formatted on individual lines. In some cases, however, it is more economical to format lower-level entries together in a single paragraph. With the titletoc package this becomes possible.

```
\titlecontents*{type}[left-indent]{before-code}{numbered-entry-format}
                    {numberless-entry-format}{page-format}[mid-code]
\titlecontents*{type}...{page-format}[mid-code][final-code]
\titlecontents*{type}...{page-format}[start-code][mid-code][final-code]
```

The `\titlecontents*` declaration is used for entries that should be formatted together with other entries of the same or lower level in a single paragraph. The first six arguments are identical to those of `\titlecontents` described on page 61.

Instead of a vertically oriented *below-code* argument, `\titlecontents*` provides one to three optional arguments that handle different situations that can happen when entries are about to be joined horizontally. All three optional arguments are by default empty. The joining works recursively as follows:

- If the current entry is the first entry to participate in joining, then its *start-code* is executed before typesetting the entry.
- Otherwise, there has been a previous entry already participating.
  - If both entries are on the same level, then the *mid-code* is inserted.

– Otherwise, if the current entry is of a lower level, then the *start-code* for it is inserted, and we recur processing the new level.
– Otherwise, the current entry is of a higher level. First, we execute for each level that has ended the *final-code* (in reverse order). Then, if the current entry is not participating in joining, we are done. Otherwise, the *mid-code* for the entry is executed, as a previous entry of the same level should already be present (assuming a hierarchically structured document).

Careful with paragraph parameters

If several levels are to be joined, then you have to specify any paragraph layout information in the *before-code* of the highest level participating. Otherwise, the scope of your settings does not include the paragraph end and thus is not applied. In the following example, `\footnotesize` applies only to the section entries — the `\baselineskip` for the whole paragraph is still set in `\normalsize`. This artificial example shows how one can join two different levels using the three optional arguments. Note in particular the spaces added at the beginning of some arguments to get the right result when joining.

The Structure of a LaTeX Document, 21 *{The overall structure of a source file; Sectioning commands; Table of contents structures; Managing references; Document source management}* • Basic Formatting Tools, 119 *{Shaping your paragraphs; Dealing with special characters; Generated or specially formatted text; Various ways of highlighting and quoting text; Footnotes, endnotes, and marginals}* ¶

```
\usepackage{titletoc,xcolor} \contentsmargin{0pt}
\titlecontents*{chapter}[0pt]
     {\sffamily}{}{}{, \thecontentspage}
     [\ \textbullet \ ] [~\P]   % mid, finish
\titlecontents*{section}[0pt]
     {\color{blue}\footnotesize\slshape}{}{}{}
     [ \{] [; ] [\}]     % start, mid, finish
\setcounter{tocdepth}{1}
\sloppy \input{partial.toc}
```

2-3-9

Let us now see how this works in practice. In the next example we join the section level, separating entries by a bullet surrounded by some stretchable space (`\xquad`) and finishing the list with a period. The chapter entries are interesting as well, because we move the page number to the left. Both types omit the heading numbers completely in this design. Because there are no page numbers at the right, we also set the right margin to zero.

21 The Structure of a LaTeX Document
The overall structure of a source file, 22 • Sectioning commands, 32 • Table of contents structures, 54 • Managing references, 75 • Document source management, 108.

119 Basic Formatting Tools
Shaping your paragraphs, 120 • Dealing with special characters, 147 • Generated or specially formatted text, 154 • Various ways of highlighting and quoting text, 177 • Footnotes, endnotes, and marginals, 204.

```
\usepackage{titletoc}
\contentsmargin{0pt}
\titlecontents{chapter}[0pt]
     {\addvspace{1.4pc}\bfseries}
     {{\Huge\thecontentspage\quad}}{}{}
\newcommand\xquad
     {\hspace{1em plus.4em minus.4em}}
\titlecontents*{section}[0pt]
     {\filright\small}{}{}
     {,~\thecontentspage}
     [\xquad\textbullet\xquad][.]
\setcounter{tocdepth}{1}
\input{partial.toc}
```

2-3-10

As a second example we look at a setup implementing a layout close to the one used in *Methods of Book Design* [198]. This design uses Garamond fonts with oldstyle digits, something we achieve by using the `garamondx` package. The `\chapter` titles are set in small capitals. To arrange that we use `\scshape` and turn all letters in the title to lowercase using `\MakeLowercase` (remember that the last token of the *numbered-entry-format* and the *numberless-entry-format* arguments can be a command with one argument to receive the heading text). The sections are all run together in a paragraph with the section number getting a § sign prepended. Separation between entries is a period followed by a space, and the final section is finished with a period as well.

Justifying the paragraph really requires a wider measure than available in the example, even though it comes out fairly well with the given text. If not, consider using `\filright`, but that would rather drastically alter the design.

```
\usepackage[osf]{garamondx}
\usepackage{titletoc}
\contentsmargin{0pt}
\titlecontents{chapter}[1.5pc]
  {\addvspace{2pc}\large}
  {\contentslabel{2pc}%
   \scshape\MakeLowercase}
  {\scshape\MakeLowercase}
  {\hfill\thecontentspage}
  [\vspace{2pt}]
\titlecontents*{section}[1.5pc]
  {\small}{\S\thecontentslabel\ }
  {}{,~\thecontentspage}[.\ ][.]
\setcounter{tocdepth}{1}
\input{partial.toc}
```

2 THE STRUCTURE OF A LaTeX DOCUMENT 21

§2.1 The overall structure of a source file, 22. §2.2 Sectioning commands, 32. §2.3 Table of contents structures, 54. §2.4 Managing references, 75. §2.5 Document source management, 108.

3 BASIC FORMATTING TOOLS 119

§3.1 Shaping your paragraphs, 120. §3.2 Dealing with special characters, 147. §3.3 Generated or specially formatted text, 154. §3.4 Various ways of highlighting and quoting text, 177. §3.5 Footnotes, endnotes, and marginals, 204.

2-3-11

Generating partial table of contents lists

It is possible to generate partial contents lists using the titletoc package like we do for every chapter in this book; it provides four commands for this purpose.

`\startcontents[`*name*`]`

A partial table of contents is started with `\startcontents`. It is possible to collect data for several partial TOCs in parallel, such as one for the current `\part` as well as one for the current `\chapter`. In that case the optional *name* argument allows us to distinguish between the two (its default value is the string `default`). Concurrently running partial TOCs are allowed to overlap each other, although normally they will be nested. All information about these partial TOCs is stored in a single file with the extension `.ptc`; this file is generated once a single `\startcontents` command is executed.

`\printcontents[`*name*`]{`*prefix*`}{`*start-level*`}{`*toc-code*`}`

This command prints the current partial TOC started earlier by `\startcontents` and includes all entries up to the next invocation of `\startcontents`. If the optional *name* argument is used, then a partial contents list with that *name* must have been started earlier.

It is quite likely that you want to format the partial TOC differently from the main table of contents. To allow for this the *prefix* argument is prepended to any entry *type* when looking for a layout definition provided via `\titlecontents` or its starred form. In the example below we used `p-` as the *prefix* and then defined a formatting for `p-subsection` to format `\subsection` entries in the partial TOC.

The *start-level* argument defines the first level that is shown in the partial TOC; in the example we used the value 2 to indicate that we want to see all subsections and lower levels.

The depth to which we want to include entries in the partial TOC can be set in *toc-code* by setting the `tocdepth` counter to a suitable value. Other initializations for typesetting the partial TOC can be made there as well. In the example we cancel any right margin, because the partial TOC is formatted as a single paragraph.

Integrating partial TOCs in the heading definitions so that there is no need to change the actual document is very easy when titletoc is used together with the titlesec package. Below we extend Example 2-2-14 from page 45 so that the `\section` command now automatically prints a partial TOC of all its subsections. This is done by using the optional *after-code* argument of the `\titleformat` declaration. We first add some vertical space, thereby ensuring that no page break can happen at this point. We next (re)start the `default` partial TOC with `\startcontents`. We then immediately typeset it using `\printcontents`; its arguments have been explained above. Finally, we set up the formatting for subsections in a partial TOC using `\titlecontents*` to run them together in a justified paragraph whose last line is centered (`\fillast`). Stringing this all together gives the desired output without any modification to the document source. Of course, a real design would also change the look and feel of the subsection headings in the document to better fit those of the sections.

A first — A longer second — An even longer fourth.

Some text to prove that this paragraph is not indented.

1.1 A first

Some text . . .

```
\usepackage{titlesec,titletoc}
\titleformat{\section}[frame]{\normalfont}
    {\footnotesize \enspace SECTION \thesection
     \enspace}{6pt}{\large\bfseries\filcenter}
    [\vspace*{5pt}\startcontents
     \printcontents{p-}{2}{\contentsmargin{0pt}}]
\titlespacing*{\section}{1pc}{*4}{*2.3}[1pc]
\titlecontents*{p-subsection}[0pt]
    {\small\itshape\fillast}{}{}{}[ --- ][.]

\section{A Title Test}
Some text to prove that this paragraph is not indented.
\subsection{A first}  Some text      \ldots \newpage
\subsection{A longer second} Some more text.
\stopcontents  \subsection{A third}  \resumecontents
\subsection{An even longer fourth}
```

2-3-12

If necessary, one can temporarily (or permanently) stop collecting entries for a partial TOC. We made use of this feature in the previous example by suppressing the third subsection.

`\stopcontents[`*name*`]`	`\resumecontents[`*name*`]`

The `\stopcontents` command stops the entry collection for the `default` partial TOC or, if used with the *name* argument, for the TOC with that *name*. At a later point the collection can be restarted using `\resumecontents`. Note that this is quite different from calling `\startcontents`, which starts a *new* partial TOC, thereby making the old entries inaccessible.

Partial TOCs do not need to be confined to a subset of your document. It is equally possible to use them to provide "overviews", e.g., listing only the chapter headings, in addition to a full table of contents. A possible implementation could look like this:

```
\AtBeginDocument{\startcontents[short]}
\newcommand\shorttoc[1]{\chapter*{#1}%
  \printcontents[short]{short-}{0}{\setcounter{tocdepth}{0}}}
\titlecontents{short-chapter}[..]{..}{..}{..}{..}
```

We start a partial contents list named `short` at the beginning of the document. Because we never restart, this partial list receives all headings. Then we define the command `\shorttoc` to produce a chapter heading without a number and then print this partial TOC list starting from level 0 (i.e., chapters) but displaying only chapters (since we set the `tocdepth` counter to zero). Finally, we define a suitable formatting for chapter entries in that list. As we used the prefix `short-`, we need to define `short-chapter` (no details given in the code above).

There are similar commands for producing partial lists of figures or tables named `\startlist`, `\printlist`, `\stoplist`, and `\resumelist` but with a slightly different syntax. For details consult the package documentation.

How we produced the content lists for this book

In this book we used these partial contents lists in several places. Each chapter starts with a `\startcontents` declaration, which enables us to show the chapter TOCs with special formatting. Each `\chapter` command executed something similar to the following:

```
\startcontents
\printcontents{p-}{1}{\contentsmargin{0pt}\setcounter{tocdepth}{1}%
                      \color{blue}\headingfont\mdseries}
```

All we had to do in addition was to provide a suitable definition for `p-section` to format the section entries. For this book we used the following setup, which was all that was necessary:

```
\titlecontents{p-section}[18pt]{\addvspace{1pt}}
              {\contentspush{\thecontentslabel\enspace}}
              {}
              {\titlerule*[6pt]{.}\ \thecontentspage}
```

Furthermore, for the overall content lists we also deployed partial lists, because both physical books have been produced in a single run (to simplify cross-referencing and indexing). We therefore started each part of the book with

```
\startcontents[part] \startlist[part]{lof} \startlist[part]{lot}
```

thereby dividing the content lists in the two parts representing the two physical books. This enabled us to automatically include the headings of both books in the table of contents for each book — with suitable formatting; i.e., in book I, we show only the chapter titles of book II, while in book II only the chapters of book I are listed, but chapters and sections are given for book II. The situation for the list of figures and tables is simpler: here we show only those entries that belong to the current book. But again this is possible only because we have divided the content lists as shown above.

2.3.3 multitoc — Setting contents lists in multiple columns

Setting contents lists in multiple columns is a design that is sometimes requested. A solution for this is provided through the multitoc package by Martin Schröder, which internally uses the multicol package to achieve the desired result.

The package has three options (`toc`, `lof`, and `lot`) to typeset the table of contents, the list of figures, or the list of tables in multiple columns (default 2).

More columns are seldom needed, but if necessary, you can specify the desired number of columns by changing `\multicolumntoc`, `\multicolumnlof`, or `\multicolumnlot` with `\renewcommand`.

2.3.4 LaTeX's low-level interfaces

In this final section on TOCs we briefly review the basic interfaces for contents files as provided by LaTeX, because you may find them used directly in older class files. Packages like titletoc also invoke them but offer some additional level of abstraction on top.

Entering information into the contents files

The interface for writing to the contents files consists of two commands: `\addcontentsline` and `\addtocontents`. They are automatically invoked by heading or caption commands, but if necessary, it is also possible to use them to enter some information directly into the files.

`\addcontentsline{`*ext*`}{`*type*`}{`*text*`}`

The `\addcontentsline` command writes the *text* together with some additional information, such as the page number of the current page, into a file with the extension *ext* (usually `.toc`, `.lof`, or `.lot`). Fragile commands within *text* need to be protected with `\protect`. The *type* argument is a string that specifies the kind of contents entry that is being made. For the table of contents (`.toc`), it is usually the name of the heading command without a backslash; for `.lof` or `.lot` files, `figure` or `table` is normally specified.

The `\addcontentsline` instruction is invoked automatically by the document sectioning commands or by the `\caption` commands within the float environments. Unfortunately, the interface has only one argument for the variable text, which makes it awkward to properly identify an object's number if present. Because such numbers (e.g., the heading number) typically need special formatting in the contents lists, this identification is absolutely necessary. The trick used by the current LaTeX

kernel to achieve this goal is to surround such a number with the command `\numberline` within the *text* argument as follows:

`\protect\numberline{`*number*`}`*heading*

For example, a `\caption` command inside a `figure` environment saves the caption text for the figure using the following line:

`\addcontentsline{lof}{figure}{\protect\numberline{\thefigure}`*caption text*`}`

Because of the `\protect` command, `\numberline` is written unchanged into the external file, while `\thefigure` is replaced along the way so that the actual figure number and not the command ends up in the file.

Later, during the formatting of the contents lists, a suitable definition of `\numberline` can then be used to format the number in a special way, such as by providing extra space or a different font. The disadvantage of this approach is that it is less general than a version that takes a separate argument for this number (e.g., you cannot easily do arbitrary transformation on this number), and it requires an appropriate definition for `\numberline` — something that is unfortunately not always easy to provide (see the discussion below).

`\addtocontents{`*ext*`}{`*text*`}`

The `\addtocontents` command does not contain a *type* parameter and is intended to enter special formatting information not directly related to any contents line. For example, the `\chapter` command of the standard classes places additional white space in the `.lof` and `.lot` files to separate entries from different chapters as follows:

```
\addtocontents{lof}{\protect\addvspace{10pt}}
\addtocontents{lot}{\protect\addvspace{10pt}}
```

By using `\addvspace` at most 10 points separate the entries from different chapters without producing strange gaps if some chapters do not contain any figures or tables.

This example, however, shows a certain danger of the interface: while `\addcontentsline`, `\addtocontents`, and `\addvspace` appear to be user-level commands (given that they do not contain any `@` signs in their names), they can easily produce strange errors.[1] In particular, `\addvspace` can be used only in vertical mode, which means that a line like the above works correctly only if an earlier `\addcontentsline` ends in vertical mode. Thus, you need to understand how such lines are actually processed to be able to enter arbitrary formatting instructions between them. This is the topic of the next section.

Potential problems with `\addvspace`

If either `\addcontentsline` or `\addtocontents` is used within the source of a document, one important restriction applies: neither command can be used at the same level as an `\include` statement. That means, for example, that the sequence

```
\addtocontents{toc}{\protect\setcounter{tocdepth}{1}}
\include{sect1}
```

with `sect1.tex` containing a `\section` command would surprisingly result in a `.toc` file containing

```
\contentsline {section}{\numberline {1}Section from sect1}{2}{}%
\setcounter {tocdepth}{1}
```

showing that the lines appear out of order. The solution is to move the `\addtocontents` or `\addcontentsline` statement into the file loaded via `\include` or to avoid `\include` altogether.

Potential problems with `\include`

Typesetting a contents list

As discussed above, contents lists are generated by implicitly or explicitly using the commands `\addcontentsline` and `\addtocontents`. The exact effect of `\addcontentsline`{*ext*}{*type*}{*text*} is to place the line

`\contentsline{`*type*`}{`*text*`}{`*page*`}{`*anchor-name*`}%`

[1] For an in-depth discussion of `\addvspace`, see Appendix A.2.4, page →II 655.

including the final percent sign into the auxiliary file with extension *ext*, where *page* is the current page number in the document. The *anchor-name* argument is by default empty but gets filled if the hyperref package is loaded. In that case it specifies a hyperlink anchor name.

The command `\addtocontents{`*ext*`}{`*text*`}` is simpler: it just puts *text* into the auxiliary file without any extra material. Thus, a typical contents list file consists of a number of `\contentsline` commands, possibly interspersed with further formatting instructions added as a result of `\addtocontents` calls. It is also possible for the user to create a table of contents by hand with the help of the command `\contentsline`.

Inconsistency with `\part`

A typical example is shown below. Note that most (though not all) heading numbers are entered as a parameter of the `\numberline` command to allow formatting with the proper indentation. For historical reasons LaTeX is unfortunately not consistent here; the standard classes do not use `\numberline` for `\part` headings but instead specify the formatting explicitly.[1]

```
\setcounter{tocdepth}{3}
\contentsline {part}{I\hspace{1em}Part}{2}{}%
\contentsline{chapter}{\numberline{1}A-Head}{2}{}%
\contentsline{section}{\numberline{1.1}B-Head}{3}{}%
\contentsline{subsection}%
    {\numberline{1.1.1}C-Head}{4}{}%
\contentsline{subsection}%
    {\numberline{}With Empty Number}{5}{}%
\contentsline{subsection}{Unnumbered C-Head}{6}{}%
\contentsline{subsection}%
    {\numberline{1.1.2}Another C-Head}{8}{}%
\contentsline{section}%
    {\numberline{1.2}Another B-Head}{10}{}%
```

2-3-13

I Part 2

1 A-Head 2

1.1 B-Head 3

1.1.1 C-Head 4

With Empty Number . 5

Unnumbered C-Head 6

1.1.2 Another C-Head . . . 8

1.2 Another B-Head 10

The `\contentsline` command is implemented to take its first argument *type* and then use it to call the corresponding `\l@`*type* command, which does the actual typesetting. One separate command for each of the types must be defined in the class file. For example, in the report class you find the following definitions:

```
\newcommand\l@section      {\@dottedtocline{1}{1.5em}{2.3em}}
\newcommand\l@subsection   {\@dottedtocline{2}{3.8em}{3.2em}}
\newcommand\l@subsubsection{\@dottedtocline{3}{7.0em}{4.1em}}
\newcommand\l@paragraph    {\@dottedtocline{4}{10em}{5em}}
\newcommand\l@subparagraph {\@dottedtocline{5}{12em}{6em}}
\newcommand\l@figure       {\@dottedtocline{1}{1.5em}{2.3em}}
\newcommand\l@table        {\l@figure}
```

By defining `\l@`*type* to call `\@dottedtocline` (a command with five arguments) and specifying three arguments (*level*, *indent*, and *numwidth*), the remaining arguments, *text* and *page*, of `\contentsline` are picked up by `\@dottedtocline` as arguments 4 and 5. The last argument (which is by default empty) is simply left sitting there doing nothing. If hyperref is loaded, the definitions are changed and the last argument is also processed.

Note that some section levels build their table of contents entries in a somewhat more complicated way so that the standard document classes have definitions for `\l@part` and `\l@chapter` (or `\l@section` with article) that do not use `\@dottedtocline`. Generally they use a set of specific formatting commands, perhaps omitting the ellipses and typesetting the title in a larger font.

So to define the layout for the contents lists, we have to declare the appropriate `\l@`*type* commands (which is precisely what titletoc's `\dottedcontents` and `\titlecontents` commands do). One easy way without this package, as shown above, is to use `\@dottedtocline`, an internal command that we will now look at in some detail.

[1] The titlesec package offers the option `newparttoc` to repair this defect.

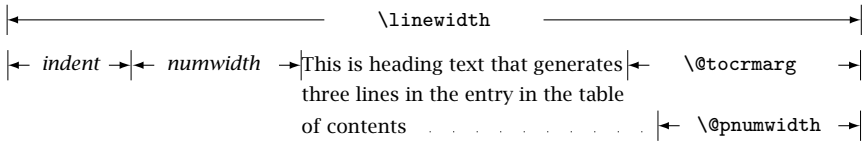


Figure 2.2: Parameters defining the layout of a contents file

`\@dottedtocline{`*level*`}{`*indent*`}{`*numwidth*`}{`*text*`}{`*page*`}`

The last two arguments of `\@dottedtocline` coincide with the second and third arguments of `\contentsline`, which itself usually invokes a `\@dottedtocline` command. The other arguments are the following:

level The nesting level of the entry. With the help of the counter `tocdepth` the user can control how many nesting levels are displayed. Levels greater than the value of this counter will not appear in the table of contents.

indent The total indentation from the left margin.

numwidth The width of the box that contains the number if *text* has a `\numberline` command. It is also the amount of extra indentation added to the second and later lines of a multiple-line entry.

Additionally, the command `\@dottedtocline` uses the following global formatting parameters, which specify the visual appearance of all entries. Although all parameters store length values, they have to be changed with `\renewcommand`!

`\@pnumwidth` The width of the box in which the page number is set.

`\@tocrmarg` The indentation of the right margin for all but the last line of multiple-line entries. It can be set to a rubber length, which results in the TOC being set unjustified.

`\@dotsep` The separation between dots, in `mu` (math units).[1] The value stored is a pure number (like `1.7` or `2`). By making this number large enough you can get rid of the dots altogether.

A pictorial representation of the effects described is shown in Figure 2.2. The field identified by *numwidth* contains a left-justified section number, if present. You can achieve the proper indentation for nested entries by varying the settings of *indent* and *numwidth*.

Problem with too many headings on one level

One case in which this is necessary, while using a standard class (`article`, `report`, or `book`), arises when you have ten or more sections and within the later ones more than nine subsections. In that case numbers and text will come too close together or even overlap if the *numwidth* argument on the corresponding calls to `\@dottedtocline` is not extended, as seen in the following example.

10 A-Head **3**
10.1 B-Head 3
...
10.9 B-Head 7
10.10B-Head 8

2-3-14

```
\contentsline{section}{\numberline{10}A-Head}{3}{}%
\contentsline{subsection}{\numberline{10.1}B-Head}{3}{}%
\ldots % several more heading lines here (not shown)
\contentsline{subsection}{\numberline{10.9}B-Head}{7}{}%
\contentsline{subsection}{\numberline{10.10}B-Head}{8}{}%
```

Redefining `\l@subsection` to leave a bit more space for the number (i.e., the third argument to `\@dottedtocline`) gives a better result in this case. You will probably have to adjust the other

[1] There are 18 `mu` units to an `em`, where the latter is taken from the `\fontdimen2` of the math symbol font `symbols`. See Section 9.8.1 on page 745 for more information about `\fontdimens`.

commands, such as `\l@subsubsection`, as well to produce a balanced look for the whole table.

10 A-Head **3**
...
10.9 B-Head 7
10.10 B-Head 8

```
\makeatletter
\renewcommand\l@subsection{\@dottedtocline{2}{1.5em}{3em}}
\makeatother
\contentsline{section}{\numberline{10}A-Head}{3}{}%
\ldots % several more heading lines here
\contentsline{subsection}{\numberline{10.9}B-Head}{7}{}%
\contentsline{subsection}{\numberline{10.10}B-Head}{8}{}%
```

2-3-15

Another example that requires changes is the use of unusual page numbering. For example, if the pages are numbered by part and formatted as "A–78", "B–328", and so on, then the space provided for the page number is probably too small, resulting at least in a large number of annoying "Overfull hbox" warnings, but more likely in some bad spacing around them. In that case the remedy is to set `\@pnumwidth` to a value that fits the widest entry — for example, via

```
\makeatletter \renewcommand\@pnumwidth{2cm} \makeatother
```

When adjusting `\@pnumwidth` this way, it is likely that the value of `\@tocrmarg` needs to be changed as well to keep the layout of the table of contents consistent.

These examples and their remedies clearly show the advantages of the higher-level interfaces provided by titletoc where commands like `\contentspush` allow for much simpler solutions.

Providing additional contents files

You may want to mark up other data in your document and display it as a list. If so, you need to create a new contents file and then make use of the facilities described above.

For example, suppose you want to collect notes on artists. For this we need to define two commands. The first command, `\artist`, typesets the artist's name and associates both of its arguments with the current position in the document by writing them and the current page number to the contents file. The second command, `\listofartistnotes`, reads the information written to the contents file on the previous run and typesets it at the point in the document where the command is called.

For this, the `\listofartistnotes` command invokes `\@starttoc{`*ext*`}`, which reads the external file (with the extension *ext*) and then reopens it for writing. This command is also used by the commands `\tableofcontents`, `\listoffigures`, and `\listoftables`. The supplementary file could be given any unused extension such as `.rec`. A command like `\chapter*{Notes on artists}` can be put in front or inside of `\listofartistnotes` to produce a title and, if desired, one can signal the presence of this list to the reader by entering it into the `.toc` file with an `\addcontentsline` command.

The actual typesetting of the individual entries in the `.rec` file is controlled by `\l@note`, which needs to be defined. In the example below, the notes are typeset as paragraphs followed by an italicized page number. Instead of defining this command directly we could have used titletoc's interfaces, e.g., `\titlecontents{note}...`

```
\newcommand\artist[2]
          {#1\addcontentsline{rec}{note}{#1: #2}}
\makeatletter \newcommand\listofartistnotes
    {\section*{Notes on artists}\@starttoc{rec}}
 \newcommand\l@note[2]
    {\par\noindent#1,~\textit{#2}\par}   \makeatother
The version of Ravel's Boléro by \artist{Jacques
Loussier Trio}{A strange experience} is rather
unusual. Quite interesting is Davis' Blue in Green
by \artist{Cassandra Wilson}{A wonderful version}.
\listofartistnotes
```

2-3-16

The version of Ravel's Boléro by Jacques Loussier Trio is rather unusual. Quite interesting is Davis' Blue in Green by Cassandra Wilson.

Notes on artists

Jacques Loussier Trio: A strange experience, *1*
Cassandra Wilson: A wonderful version, *1*

The float package described in Section 7.3.1 on page 529 implements the above mechanism with the command `\listof`, which generates a list of floats of the type specified as its argument.

2.4 Managing references

LaTeX has commands that make it easy to manage references in a document. In particular, it supports *cross-references* (internal references between elements within a document), *bibliographic* citations (references to external documents), and *indexing* of selected words or expressions. Indexing facilities will be discussed in Chapter 14, and bibliographic citations in Chapters 15 and 16.

To allow cross-referencing of elements inside a document, you should assign a "key" (consisting of a string of characters, preferably ASCII letters, digits, and punctuation) to the given structural element and then use that key to refer to that element elsewhere.

`\label{`*key*`}` `\ref{`*key*`}` `\pageref{`*key*`}`

The `\label` command assigns the *key* to the currently "active" element of the document (see below for determining which element is active at a given point). The `\ref` command typesets a string, identifying the given element — such as the section, equation, or figure number — depending on the type of structural element that was active when the `\label` command was issued. The `\pageref` command typesets the number of the page where the `\label` command was given. The *key* strings should, of course, be unique. As a simple aid it can be useful to prefix them with a string identifying the structural element in question: `sec` might represent sectional units, `fig` would identify figures, and so on.

```
\section{A Section} \label{sec:this}

A reference to this section looks
like this:  ``see section~\ref{sec:this}
on page~\pageref{sec:this}''.
```

4 A Section

A reference to this section looks like this: "see section 4 on page 6".

2-4-1

There is a potential danger when using punctuation characters such as a colon. In certain language styles within the `babel` system (see Chapter 13), some of these characters have special meanings and behave essentially like commands. The `babel` package tries hard to allow such characters as part of `\label` keys, but this can fail in some situations. Similarly, characters outside the ASCII range have been a problem in the past. However, starting with the LaTeX release in 2019 there is a new implementation that essentially supports all Unicode characters that can also be used for typesetting text, i.e., are not generally rejected because LaTeX does not know how to deal with them. Thus, you can use labels like "`fig:größer`", but using, say, Chinese characters may still give you errors, unless you have loaded special font support packages for them or used a fairly recent LaTeX release.[1]

Restrictions on the characters used in keys

For building cross-reference labels, the "currently active" structural element of a document is determined in the following way. The sectioning commands (`\chapter`,

[1] With real Unicode engines, such as XeTeX or LuaTeX, all Unicode characters are usable. The remaining technical restrictions of the pdfTeX engine were finally overcome with the November 2021 release of LaTeX — so now you can also use all Unicode characters with that engine.

\section, ...), the environments equation, figure, table, and the theorem family, as well as the various levels of the enumerate environment, and \footnote set the *current reference string*, which contains the number generated by LaTeX for the given element. This reference string is usually set at the beginning of an element and reset when the scope of the element is exited.

Problems with wrong references to floats

Notable exceptions to this rule are the table and figure environments, where the reference string is defined by the \caption commands. This allows several \caption and \label pairs inside one environment.[1] Because it is the \caption directive that generates the number, the corresponding \label command must *follow* the \caption command in question. Otherwise, an incorrect number is generated. If placed earlier in the float body, the \label command picks up the *current reference string* from some earlier entity, typically the current sectional unit.

The problem is shown clearly in the following example, where only the labels "fig:in2" and "fig:in3" are placed correctly to generate the needed reference numbers for the figures. In the case of "fig:in4" it is seen that environments (in this case, center) limit the scope of references, because we obtain the number of the current section, rather than the number of the figure.

Do not use center in floats

It should be noted that using a center environment in a float (like we did below) is not a good idea not just because it limits the reference scope: it also creates a usually unwanted extra space at the top of the float! It is better to use a \centering declaration, which avoids both problems.

```
\section{A section}
\subsection{A subsection}\label{sec:before}
Text before is referenced as ‘\ref{sec:before}’.

\begin{figure}[ht]           \label{fig:in1}   % bad
 \begin{center}
   \fbox{\ldots{} figure body \ldots}
   \caption{First caption}   \label{fig:in2}   % ok
   \bigskip
   \fbox{\ldots{} figure body \ldots}
   \caption{Second caption}  \label{fig:in3}   % ok
 \end{center}                \label{fig:in4}   % bad
\end{figure}
\label{sec:after} % bad, unless you want the page reference

\raggedright
The labels are: ‘before’ (\ref{sec:before}),
‘fig:in1’ (\ref{fig:in1}) -- bad, ‘fig:in2’ (\ref{fig:in2}),
‘fig:in3’ (\ref{fig:in3}), ‘fig:in4’ (\ref{fig:in4}) -- bad
and ‘after’ (\ref{sec:after})  -- probably bad!
```

2-4-2

3 A section

3.1 A subsection

Text before is referenced as ‘3.1’.

… figure body …

Figure 1: First caption

… figure body …

Figure 2: Second caption

The labels are: ‘before’ (3.1), ‘fig:in1’ (3.1) – bad, ‘fig:in2’ (1), ‘fig:in3’ (2), ‘fig:in4’ (3.1) – bad and ‘after’ (3.1) – probably bad!

[1] There are, however, good reasons for not placing more than one \caption command within a float environment. Typically proper spacing is difficult to achieve, and, more importantly, it limits LaTeXs options to place the float and should (if at all) be done only during final layout adjustments.

For each *key* declared with `\label{`*key*`}`, LaTeX records the current reference string and the page number. Thus, multiple `\label` commands (with different key identifiers *key*) inside the same sectional unit generate an identical reference string but, possibly, different page numbers like `sec:before` and `sec:after` above.


Label commands inside arguments

According to the *LaTeX Manual* [106] labels can be placed inside the main argument of heading or caption commands, rather than after them. Doing this makes the source a little less readable (which is why I prefer them after), but there are some edge cases, usually with `\caption`, where placing the label after the command can result in some incorrect extra space, so you need to watch out for this.

Fancier labels

A reference via `\ref` produces, by default, the data associated with the corresponding `\label` command (typically a number); any additional formatting must be provided by the user. If, for example, references to equations are always to be typeset as "equation (*number*)", one has to code "`equation (\ref{`*key*`})`".

To enforce consistency the amsmath package provides an `\eqref` command to reference equations. It automatically places parentheses around the equation number. To utilize this and also get varioref's magic applied (see next section), one could define

```
\newcommand\eqvref[1]{\eqref{#1} \vpageref{#1}}
```

which then automatically adds a page reference if the equation is on a different page. What that does not do is to automatically add the word "`equation`", though you could, of course, code that into the definition as well. However, a more general solution for adding words based on the referenced counter is offered with the `\labelformat` declaration. Alternatively you can use the cleveref package discussed in Section 2.4.2, which provides a more sophisticated solution for this.

`\labelformat{`*counter*`}{`*formatting-code*`}` `\Ref{`*label*`}`

With `\labelformat` LaTeX offers a possibility to generate such frills automatically.[1] The command takes two arguments: the name of a counter and its representation when referenced. Thus, for a successful usage, one has to know the counter name being used for generating the label, though in practice this should not pose a problem. When processing a reference the current counter number (or, more exactly, its representation) is picked up as an argument, so the second argument should contain `#1` to retrieve it.

A side effect of using `\labelformat` is that, depending on the defined formatting, it becomes impossible to use `\ref` at the beginning of a sentence (if its replacement text starts with a lowercase letter). To overcome this problem there is also a `\Ref` command that behaves like `\ref` except that it uppercases the first token

[1] In the past this command was provided by the varioref package.

of the generated string. In the following example, you can observe this behavior when "section" is turned into "Section".

1 An example

Section 1 shows the use of the \labelformat declaration with a reference to equation (1).

$$a = b \tag{1}$$

```
\usepackage[nospace]{varioref}
\labelformat{section}{section~#1}
\labelformat{equation}{equation~(#1)}
\section{An example}\label{sec}
\Ref{sec} shows the use of the \verb=\labelformat=
declaration with a reference to \ref{eq}.
\begin{equation} a = b \label{eq} \end{equation}
```

2-4-3

To make the \Ref command work properly, the first token in the second argument of \labelformat has to be a single ASCII letter; otherwise, the capitalization fails or, even worse, you end up with some error messages. If you actually need something more complicated in this place (e.g., an accented letter), you have to explicitly surround it with braces, thereby identifying the part that needs to be capitalized. For example, for figure references in the Hungarian language you might want to write \labelformat{figure}{{á}bra~\thefigure}.

Unicode engines

In pdfTeX the braces are necessary, regardless of whether you write the accented character as \'a or as á as we did above, because in this engine UTF-8 characters are seen as several tokens even if on the screen they look like a single character. The downside is that these braces prevent any kerning that the font may specify between á and the following character. However, in XeTeX or LuaTeX a Unicode character is a single token (not a sequence of bytes) and is therefore picked up correctly even without the braces. Thus, with these engines the braces should not be used to improve the typeset result.

As a second example of the use of \labelformat consider the following situation: in the report or book document class, footnotes are numbered per chapter. Referencing them would normally be ambiguous, given that it is not clear whether we refer to a footnote in the current chapter or to a footnote from a different chapter. This ambiguity can be resolved by always adding the chapter information in the reference or by comparing the number of the chapter in which the \label occurred with the current chapter number and adding extra information if they differ. This is achieved by the following code:

```
\usepackage{ifthen,varioref}
\labelformat{footnote}{#1\protect\iscurrentchapter{\thechapter}}
\newcommand\iscurrentchapter[1]{%
   \ifthenelse{\equal{#1}{\thechapter}}{}{ in Chapter~#1}}
```

The trick is to use \protect to prevent \iscurrentchapter from being evaluated when the label is formed. Then, when the \ref command is executed, \iscurrentchapter compares its argument (i.e., the chapter number current when the label was formed) to the now current chapter number and, when they differ, typesets the appropriate information.

2.4.1 varioref — More flexible cross-references

In many cases it is helpful, when referring to a figure or table, to put both a `\ref` and a `\pageref` command into the document, especially when one or more pages separate the reference and the object. Some people use a command like

```
\newcommand\fullref[1]{\ref{#1} on page~\pageref{#1}}
```

to reduce the number of keystrokes necessary to make such a complete reference. But because one never knows with certainty where the referenced object finally falls, this method can result in a citation to the current page, which is disturbing and should therefore be avoided. The package varioref, written by Frank Mittelbach, tries to resolve that problem automatically. For this it provides the commands `\vref` and `\vpageref` to deal with single references, as well as `\vrefrange` and `\vpagerefrange` to handle multiple references.[1]

We recommend that you always load the package with the option `nospace`, and this is what we assume throughout the book. Without it varioref manipulates the spaces in front of its commands (and even adds one if there is not any), but this causes a number of problems and should therefore be avoided.[2] Some more details are given on page 85.

We recommend to always use the `nospace` *option*

`\vref*`[*same-page*]{*key*}	`\Vref*`[*same-page*]{*key*}

The command `\vref` is like `\ref` when the reference and `\label` are on the same page and the optional argument is not used. With the optional argument it prints the text *same-page* after the reference.[3] If the label and reference differ by one page, `\vref` creates one of these strings: “on the facing page”, “on the preceding page”, or “on the following page”. The word “facing” is used when both label and reference fall on a double spread and the document is typeset in `twoside` mode. When the difference is larger than one page, `\vref` produces both `\ref` and `\pageref`. Note that when a special page numbering scheme is used instead of the usual arabic numbering (for example, `\pagenumbering{roman}`), there will be no distinction between being one or many pages off.

If `\varioref` is loaded with the option `nospace` as recommended, then the star form has no effect unless you also load hyperref. In the latter case it prevents hyperref from generating a hyperlink for this reference. If `nospace` is not used, then the star form stops adding a space in front of the reference.

Different behaviors of the star form depending on options used

The `\Vref` command works like `\vref` except that it internally uses `\Ref` instead of `\ref`; i.e., it uppercases the first letter. See above for a discussion of the restrictions that apply to its use with pdfTEX.

[1] As a matter of fact, the package also defines `\fullref` for cases where it is certain that label and reference are far apart. Using that instead of `\vref` needs less resources and is faster although these days this seldom matters.

[2] The reason that `nospace` is not the default is that the documents in the last twenty years assumed the old behavior, and thus changing the default would break too many documents out there.

[3] Note that the optional arguments of `\vref`, `\vpageref`, and similar commands from varioref are not supported if you also load the cleveref package! See Section 2.4.2 on page 86 for the restrictions.

`\vpageref*[`*same-page*`][`*other-page*`]{`*key*`}`

Sometimes you may only want to refer to a page number. In that case, a reference should be suppressed if you are citing the current page. For this purpose the `\vpageref` command is defined. It produces the same strings as `\vref` except that it does not start with `\ref`, and it produces the string saved in `\reftextcurrent` if both label and reference fall on the same page.

Defining `\reftextcurrent` to produce something like "on the current page" ensures that text like "`... see the diagram \vpageref{ex:foo} which shows ...`" does not come out as "... see the diagram which shows ...", which could be misleading.

A space in front of `\vpageref` is ignored if the command does not create any text at all. Thus the correct way to use the command is to place a space on either side. As with `\vref` the star form has no effect when the option `nospace` is used unless hyperref is also loaded in which case it suppresses the hyperlink to the page.

In fact, `\vpageref` allows even more control when used with its two optional arguments. The first argument specifies an alternative text to be used if the label and reference fall on the same page. This is helpful when both are close together so that they may or may not be separated by a page break. In such a case, you usually know whether the reference comes before or after the label so that you can code something like the following:

```
... see the diagram \vpageref[above]{ex:foo} which shows ...
```

The resultant text will be "... see the diagram above which shows ..." when both are on the same page, or "... see the diagram on the page before which shows ..." (or something similar, depending on the settings of the `\reftext..before` and `\reftext..after` commands) if they are separated by a page break. Note, however, that if you use `\vpageref` with such an optional argument to refer to a figure or table, depending on the float placement parameters, the float may show up at the top of the current page and therefore before the reference, even if it follows the reference in the source file.[1]

Maybe you even prefer to say "... see the above diagram" when both diagram and reference fall on the same page — that is, reverse the word order compared to our previous example. In fact, in some languages the word order automatically changes in that case. To allow for this variation the second optional argument *other-page* can be used. It specifies the text preceding the generated reference if both object and reference do not fall on the same page. Thus, one would write

```
... see the \vpageref[above diagram][diagram]{ex:foo} which shows ...
```

to achieve the desired effect.

[1] To ensure that a floating object always follows its place in the source, use the flafter package, which is described in Section 7.2.

`\vpagerefrange*[`*same-page*`]{`*first*`}{`*last*`}`

This command is similar to `\vpageref` (without the second optional argument) but takes two mandatory arguments — two labels denoting a range. If both labels fall on the same page, the command acts exactly like `\vpageref` (with a single label); otherwise, it produces something like "on pages 15–18" (see the customization possibilities described below). It has an optional argument that defaults to the string stored in `\reftextcurrent` and is used if both labels appear on the current page.

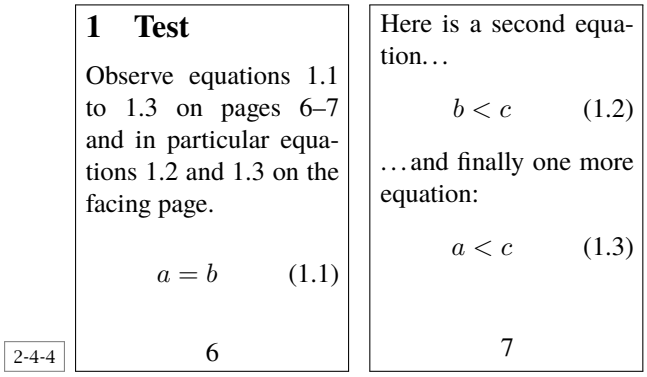
Again there exists a starred form, `\vpagerefrange*`, which suppresses a hyperlink or the insertion of a space depending on the options used.

`\vrefrange[`*same-page*`]{`*first*`}{`*last*`}`

This `\vrefrange` command is simply a convenient shorthand for

`\ref{`*first*`} to \ref{`*last*`} \vrefpagerange[`*same-page*`]{`*first*`}{`*last*`}`

except that it varies the word "to" depending on the language. This means it is suitable only for ranges of length three or more, because with just two you better use "and" between the references as we did in the following example.



1 Test

Observe equations 1.1 to 1.3 on pages 6–7 and in particular equations 1.2 and 1.3 on the facing page.

$$a = b \qquad (1.1)$$

6

Here is a second equation…

$$b < c \qquad (1.2)$$

…and finally one more equation:

$$a < c \qquad (1.3)$$

7

2-4-4

```
\usepackage[nospace]{varioref}
\renewcommand\theequation
     {\thesection.\arabic{equation}}
\section{Test}
Observe equations~\vrefrange{A}{C} and
in particular equations~\ref{B}
and~\ref{C} \vpagerefrange{B}{C}.
\begin{equation}a=b\label{A}\end{equation}
Here is a second equation\ldots
\begin{equation}b<c\label{B}\end{equation}
\ldots and finally one more equation:
\begin{equation}a<c\label{C}\end{equation}
```

Providing your own reference commands

Sometimes you may want to define your own reference commands that make use of the varioref features internally. For this the package offers three helper commands.

`\vpagerefcompare{`*key*$_1$`}{`*key*$_2$`}{`*true-code*`}{`*false-code*`}`

This command compares the page numbers for *key*$_1$ and key$_2$ and then executes either *true-code* or *false-code* depending on the result. The next example shows a not very serious application that compares two equation labels and prints out text

depending on their relative positions. Compare the results of the tests on the first page with those on the second.

Test: the equations (1) and (2) on this page.
Test: the equation (1) on the current page and (3) on page 8.

$$a = b \quad (1)$$

$$b = c \quad (2)$$

6

Test: the equations (1) and (2) on the preceding page.
Test: the equation (1) on the facing page and (3) on the next page.
We force eq. 3 to the next page!

7

```
\usepackage[nospace]{varioref}
\newcommand\veqns[2]{the equation%
  \vpagerefcompare{#1}{#2}%
    {s (\ref{#1})}%
    {  (\ref{#1}) \vpageref{#1}}%
  \space and (\ref{#2}) \vpageref{#2}}
Test: \veqns{A}{B}. \par Test: \veqns{A}{C}.
\begin{equation} a=b \label{A}\end{equation}
\begin{equation} b=c \label{B}\end{equation}
\newpage
Test: \veqns{A}{B}. \par Test: \veqns{A}{C}.
\par  We force eq.~\ref{C} to the next page!
\newpage % for eq. to next page
\begin{equation} c=a \label{C}\end{equation}
```

2-4-5

`\vpagerefnearby{`*key*`}{`*true-code*`}{`*false-code*`}`

This command lets you find out if a page reference would generate textual reference because it is on the previous, current, or next page or if it would just generate reference with a page number. Depending on the result, either the *true-code* or the *false-code* is executed.

`\vrefpagenum{`*cmd*`}{`*key*`}`

The package also provides the `\vrefpagenum` command, which allows you to write your own small commands that implement functions similar to those provided by the two previous commands. It takes two arguments: the second is a label (i.e., as used in `\label` or `\ref`), and the first is an arbitrary command name (make sure you use your own) that is set to the page number representation related to this label. This can then be used for comparisons with page numbers of other labels, but note that it may not be a number.

Language options

The package supports the options defined by the babel system (see Section 13.1.3); thus, a declaration like `\usepackage[ngerman]{varioref}` produces texts suitable for the German language. If your document is written in several languages, you need to specify all of them as options so that the strings get integrated into babel's language switching mechanism. For languages not (yet) supported you need to specify the relevant language strings yourself as explained on page 84.

Individual customizations

How to say before ...

To allow further customization, the generated text strings (which will be predefined by the language options) are all defined via macros. Backward references

use \reftextbefore if the label is on the preceding page but invisible, and \reftextfacebefore if it is on the facing page (that is, if the current page number is odd and the document is set in twoside mode).

… and after …

Similarly, \reftextafter is used when the label comes on the next page but one has to turn the page, and \reftextfaceafter is used when it is on the next, but facing, page. These four strings can be redefined with \renewcommand.

In fact, \reftextfacebefore and \reftextfaceafter are used only if the user or the document class specified two-sided printing.

… or far away

The command \reftextfaraway is used when the label and reference differ by more than one page or when they are nonnumeric. This macro is a bit different from the preceding ones because it takes one argument, the symbolic reference string, so that you can make use of \pageref in its replacement text. For instance, if you wanted to use your macros in German language documents, you would define something like:

```
\renewcommand\reftextfaraway[1]{auf Seite~\pageref{#1}}
```

Denoting ranges

The \reftextpagerange command takes two arguments and produces the text that describes a page range (the arguments are keys to be used with \pageref). Similarly, \reftextlabelrange takes two arguments and describes the range of figures, tables, or whatever the labels refer to. See below for the English language defaults of both.

Minor randomness

To allow some random variation in the generated strings, you can use the command \reftextvario inside the string macros. This command takes two arguments and selects one or the other for printing depending on the number of \vref or \vpageref commands already encountered in the document (alternating between the first and the second argument).

As an example, the English language default definitions of the various macros described in this section are shown below:

```
\newcommand\reftextfaceafter{on the \reftextvario{facing}{next} page}
\newcommand\reftextfacebefore
                      {on the \reftextvario{facing}{preceding} page}
\newcommand\reftextafter {on the \reftextvario{following}{next} page}
\newcommand\reftextbefore
                  {on the \reftextvario{preceding page}{page before}}
\newcommand\reftextcurrent {on \reftextvario{this}{the current} page}
\newcommand\reftextfaraway   [1]{on page~\pageref{#1}}
\newcommand\reftextpagerange [2]{on pages~\pageref{#1}--\pageref{#2}}
\newcommand\reftextlabelrange[2]{\ref{#1} to~\ref{#2}}
```

If you want to customize the package according to your own preferences, just write appropriate redefinitions of the above commands into the preamble of your document or in a file with the extension .sty (e.g., vrflocal.sty) and load that with \usepackage. If you also put \RequirePackage[nospace]{varioref} (see Section A.6 on page →II 693) at the beginning of this file, then your local package automatically loads the varioref package.

Using varioref *without textual references*

Some people do not like textual references to pages but want to automatically suppress a page reference when both label and reference fall on the same page. This can be achieved with the help of the `\thevpagerefnum` command as follows:

```
\renewcommand\reftextfaceafter {on page~\thevpagerefnum}
\renewcommand\reftextfacebefore{on page~\thevpagerefnum}
\renewcommand\reftextafter     {on page~\thevpagerefnum}
\renewcommand\reftextbefore    {on page~\thevpagerefnum}
```

Within one of the `\reftext...` commands, `\thevpagerefnum` evaluates to the current page number if known or to two question marks otherwise.

In the same fashion you can suppress all textual page references if the reference is on the preceding or following page and show the page number only when it is further away. For this, change the definitions as follows:

```
\renewcommand\reftextfaceafter {\unskip}
\renewcommand\reftextafter     {\unskip}
\renewcommand\reftextfacebefore{\unskip}
\renewcommand\reftextbefore    {\unskip}
```

The `\unskip` is necessary in order to remove the space that was already added after the reference. Without it you end up with two spaces.

Altering the phrase structure

Some languages have a completely different sentence structure so that adjusting only the individual phrases is not enough. To cater for this, there are also `\vrefformat`, `\Vrefformat`, `\vrefrangeformat`, and `\fullrefformat`. For example, for Japanese there are definitions such as

```
\renewcommand\vrefformat[2]{\ref{#2}(\vpageref[#1]{#2})} % for Japanese
\renewcommand\vrefformat[2]{\ref{#2} \vpageref[#1]{#2}}  % all other
                                                         % languages
```

The parentheses in the Japanese definition are not the normal characters but their full wide counterparts in Unicode slots U+FF08 and U+FF09 — something you cannot see here but is important when this is used together with Kanji glyphs.

Customization for several languages with babel

If you use the babel system, redefinitions for individual languages should be added using `\addto`, as explained in Section 13.6, e.g.,

```
\addto\extrasngerman{%
   \renewcommand\reftextfaceafter{auf der nächsten Seite}%
    ... }
```

Do not forget to add appropriate `%` signs as shown above. Otherwise, a language switch might generate spurious spaces in your document!

A few things to watch out for

Defining commands like the ones described above poses some interesting problems. Suppose, for example, that a generated text like “on the next page” gets broken across pages. If this happens, it is very difficult to find an acceptable algorithmic solution, and, in fact, this situation can even result in a document that always changes from one state to another (i.e., inserting one string; finding that this is wrong; inserting another string on the next run which makes the first string correct again; inserting ...). The current implementation of the package varioref considers the end of the generated string as being relevant. For example,


Impossible documents!

> Table 5 on the current ⟨*page break*⟩ page

would be true if Table 5 were on the page containing the word “page”, not the one containing the word “current”. However, this behavior is not completely satisfactory and in some cases may actually result in a possible loop (where LaTeX is requesting an additional run over and over again). Therefore, all such situations produce a LaTeX error message so that you can inspect the problem and perhaps decide to use a `\ref` command in that place.

During document preparation, while one is still changing the text, such errors can be turned into warnings by placing a `\vrefwarning` command in the preamble. This is equivalent to specifying `draft` as an option to the package. `\vrefshowerrors` ensures that varioref stops when detecting a possible loop. This is the default and equivalent to specifying `final` as an option. The commands can also be used inside the document if you want to disable the errors only in some places.


Variation can be dangerous!

Also, be aware of the potential problems that can result from the use of `\reftextvario` in the default definitions: if you reference the same object several times in nearby places, the change in wording every second time can look strange. To get rid of the variations introduced by `\reftextvario` without redefining all the `\reftext...` commands that use it, you can simply redefine it to always use the first or the second of its arguments, e.g.,

```
\renewcommand\reftextvario[2]{#1}
```

in the preamble of your document.

Package behavior without the `nospace` option

When varioref was originally designed, it had a special behavior: its commands removed any preceding space and inserted their own instead. Thus, you could leave out space before `\vref` or `\vpageref` and it would still put the reference in the right place. But this meant that you could not write something like `(\vref{foo})`, and therefore the package offered star forms of the commands to prevent the space manipulations. This is still the default behavior if you use the package without the `nospace` option.

However, this approach has several drawbacks. For one it prevents hyperref from using the star forms for hyperlink suppression (which is an important feature), it makes your sources less readable if you leave out the space, and it does not work

well with other packages, e.g., cleveref. This is why these days we recommend using always the nospace option.

2.4.2 cleveref — Cleverly formatted references

We have already seen on page 77 that LaTeX offers some light-weight support for formatted references based on the counter used in the reference. The package cleveref by Toby Cubitt is the heavy-weight version of this approach. In addition, it supports references to multiple labels and with numerical references or page references sorts the results and compresses ranges appropriately. The varioref commands `\vref`, `\Vref`, and `\vpageref` are augmented to support multiple keys and reference formatting.[1] All aspects of the formatting are customizable in the document preamble, which makes cleveref a truly comprehensive and powerful solution.

`\cref*{`*key-list*`}` `\Cref*{`*key-list*`}`

The main command offered by cleveref is `\cref`. It accepts either a single key (like `\ref`) or a list of such keys separated by commas. It then formats the corresponding reference (or references) according to their type, e.g., prepends words such as "section" or abbreviations such as "fig." and possibly adds other frills such as parentheses around equation numbers.

If a comma-separated list of keys is given, it uses plural forms as appropriate and in longer lists it knows about appropriate conjunctions; e.g., it can distinguish pairs, longer sets of individual references, and consecutive ranges, and it can handle combinations thereof.

Because the generated text might start with a lowercase letter, the package additionally offers `\Cref` to be used at the start of a sentence. It differs from `\cref` by using a capital first letter in the text that is prepended to the reference number. It also always uses full words, e.g., "Figure" not "Fig.", whereas `\cref` may produce abbreviations if so directed.

If the hyperref package is used, then the typeset reference gets a hyperlink to the reference target by default. Use the star form to suppress this link.

4 A Section

A reference to an equation in this section looks like: "see eq. (1) in section 4".

$$a = b \tag{1}$$
$$b < c \tag{2}$$
$$c < d \tag{3}$$

Equations (1) to (3) above are …

```
\usepackage{amsmath,cleveref}

\section{A Section}\label{sec:this}

A reference to an equation in this section looks
like: ``see \cref{eq:a} in \cref{sec:this}''.
\begin{align} a &= b \label{eq:a} \\
              b &< c \label{eq:b} \\
              c &< d \label{eq:c} \end{align}
\Cref{eq:c,eq:a,eq:b} above are \ldots
```

2-4-6

[1] The cleveref package requires varioref to be loaded with the option `nospace`, to be able to use the star forms for suppressing hyperlinks. If necessary, it enforces this varioref behavior.

As you can see, the reference to the equation is handled quite differently from the one to the heading: it uses an abbreviation and adds parentheses around the equation number, whereas the heading is referred to as “section”. In comparison, `\Cref` used “Equations”; the references are correctly sorted (even though they are given in a different order in the source), and the resulting range was correctly compressed. Automatic sorting of references is usually helpful. If you rearrange parts of your text, then some of your reference may change their order, and without this sorting, you might end up with strange references such as “see figures (1), (3), and (2)”.

If we had two additional equations and referenced some of them, the result would come out quite different as shown in the next example:

2-4-7 Equations (1) to (5) are sorted and eqs. (1) to (3) and (5) are sorted with a gap. But compare these results with referencing eqs. (1) to (3), (4) and (5)! Surprised?

```
% equations as before + 2 more
\Cref{eq:c,eq:b,eq:a,eq:d,eq:e} are sorted and
\cref{eq:c,eq:b,eq:a,eq:e} are sorted with a gap.
But compare these results with referencing
\cref{eq:c,,eq:b,eq:a,eq:d,eq:e}! Surprised?
```

The behavior of the last `\cref` in the previous example may have been a bit of a surprise: the references are correctly sorted but split into two groups with the first one compressed. The reason is the “`,,`”. It tells cleveref that the preceding key (`eq:c`) should be treated as a final reference in whatever range it belongs to after sorting. Thus, equations `eq:d` and `eq:e` form a second range or rather a pair and we therefore get this particular result in the second sentence of the example. This facility can be sometimes helpful, but in such a case you would probably want to make sure that you keep the keys sorted in the source to better understand what is going on.

If you use `\cref` or `\Cref` with a list of keys, it is not required that they are all of the same type as cleveref happily sorts them within each type and then applies the rest of its magic. Of course, this works well only if the types are compatible with each other, e.g., if you are referring to a number of different heading levels, to floats, or to different types of theorem environments, etc. Otherwise, you might end up with strange constructs.

2-4-8 In figs. 1 to 3 and table 1 we …

```
\usepackage{cleveref}
In \cref{fig:a,tab:a,fig:b,fig:c} we \ldots
```

Options to alter the package behavior

You may not fancy all of the defaults that cleveref applies, so to alter them you can use the options `sort` (but do not compress), `compress` (but do not sort), `nosort` (do neither), or `sort&compress` (the default). If the generated texts should always be capitalized, which is often requested in house styles, use the option `capitalize`.

The package also understands most language options; e.g., in the next example we use German text and turn off compression but keep the sorting. We do not have to use `capitalize`, because German nouns are always capitalized.

2-4-9 Gleichungen (1), (2) und (3) in Abschnitt 4 …

```
\usepackage[ngerman,sort]{cleveref}
\Cref{eq:c,eq:a,eq:b} in \cref{sec:this} \ldots
```

Another useful option is `noabbrev` if you do not like the abbreviations used by `\cref` in some languages such as English.

Finally, if you use cleveref with hyperref, then references are hyperlinked to their target (unless the star forms of the commands are used). By default the link area, i.e., the text you can click to navigate, is only the label and does not include the additional material. With the option `nameinlink`, you can change this to enlarge the clickable area. To demonstrate this we colored the link areas in the next example:

```
\usepackage[colorlinks,linkcolor=blue]{hyperref}
\usepackage[nameinlink,noabbrev]{cleveref}
\Cref{sec:this} contains \cref{eq:a,eq:b}.
```

Section 4 contains equations (1) and (2). 2-4-10

`\namecref{`*key*`}`	`\nameCref{`*key*`}`	`\lcnamecref{`*key*`}`
`\namecrefs{`*key*`}`	`\nameCrefs{`*key*`}`	`\lcnamecrefs{`*key*`}`

Sometimes it is useful to provide just the text generated for a certain reference type without typesetting the label value. The above commands do this for use within a sentence and at the start of a sentence, both in singular and plural forms. The `\lcname...` commands always use lowercase, even if the `capitalize` option is in force. All of the commands accept only a single *key* as their argument, because a key list would be pointless if no labels are set.

`\labelcref{`*key-list*`}`	`\labelcpageref{`*key-list*`}`

There are also `\labelcref` and `\labelcpageref` that print the labels or page references without prepending any text. They support *key-list*s and still add any necessary conjunction text between the items. However, because no text denoting the type is typeset, the elements in the *key-list* must be of a single type.

`\crefrange*{`key_1`}{`key_2`}`	`\Crefrange*{`key_{first}`}{`key_{last}`}`

Instead of specifying a lengthy *key-list* with `\cref`, you can use `\crefrange` or `\Crefrange` using the *first* and *last* keys to denote a consecutive range. Note that the assumption is that this range has at least three items; thus, referencing a range of length two comes out slightly strange as shown below. For this you therefore should use `\cref{eq:b,eq:c}`.

```
\usepackage{cleveref}
\Crefrange{eq:a}{eq:e} and in particular
\crefrange{eq:b}{eq:c} show \ldots
```

Equations (1) to (5) and in particular eqs. (2) to (3) show … 2-4-11

`\cpageref{`*key-list*`}`	`\Cpageref{`*key-list*`}`
`\cpagerefrange{`key_{first}`}{`key_{last}`}`	`\Cpagerefrange{`key_{first}`}{`key_{last}`}`

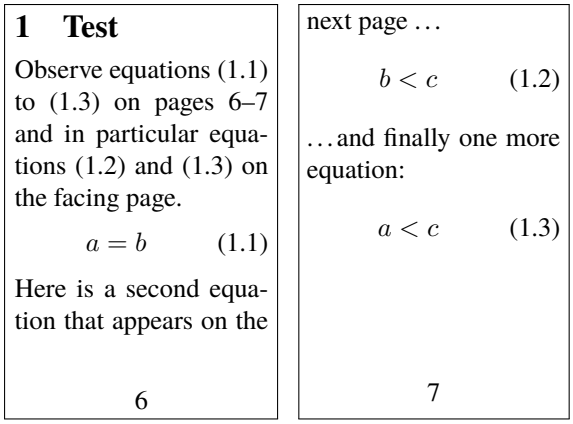
These are the commands to deal with references to page number and, just like with `\cref`, sort and compress them and add the appropriate words and punctuations in the target language.

`\vref*{`*key-list*`}`	`\Vref*{`*key-list*`}`	`\vrefrange*{`key_{first}`}{`key_{last}`}`
`\vpageref*{`*key-list*`}`		`\vpagerefrange*{`key_{first}`}{`key_{last}`}`

If the varioref package is used with cleveref, then some of its functions are changed to support *key-lists* instead of only a single *key* as arguments. Note that optional arguments are not supported if both packages are used together. For use at the beginning of a sentence cleveref also defines `\Vrefrange`, `\Vpageref`, and `\Vpagerefrange`, which are not offered by varioref.

Below we repeat Example 2-4-4 on page 81 with both packages loaded. Note that we can now simply use `\vref` with a *key-list* instead of the construction used before.

2-4-12



1 Test

Observe equations (1.1) to (1.3) on pages 6–7 and in particular equations (1.2) and (1.3) on the facing page.

$$a = b \qquad (1.1)$$

Here is a second equation that appears on the

6

next page ...

$$b < c \qquad (1.2)$$

... and finally one more equation:

$$a < c \qquad (1.3)$$

7

```
\usepackage[nospace]{varioref}
\usepackage[noabbrev]{cleveref}
\renewcommand\theequation
     {\thesection.\arabic{equation}}
\section{Test}
Observe \vrefrange{A}{C} and
in particular \vref{B,C}.
\begin{equation}a=b\label{A}\end{equation}
Here is a second equation that appears
on the next page \ldots
\begin{equation}b<c\label{B}\end{equation}
\ldots and finally one more equation:
\begin{equation}a<c\label{C}\end{equation}
```

Customizing the references

The text generated by the cleveref commands depend on the "type" of the reference, which is usually based on the counter used by the reference.[1] For example, `\section` commands use the `section` counter, `figure` environments the `figure` counter, `enumerate` the counters `enumi` to `enumiv` for its different nesting levels, and so on. Thus, the reference type for a second-level enumeration is `enumii`, while that to a figure is `figure`. There are a few exceptions to the rule: the heading levels in the back matter have the types `appendix`, `subappendix`, etc., and theorem-like environments use the environment name if amsthm or ntheorem is loaded.

As the package has knowledge about all these standard types and defines default texts for them, it can be used out of the box generating results like those shown in the previous examples.

However, if you load additional packages that define their own environments or commands with referenceable counters or if you simply do not like the default texts generated by cleveref, then it is easy to adjust or extend them using the configuration possibilities offered by the package as discussed below.

[1] As a side effect this means that if two different environments use the same counter, then references to them are of the same type and thus always generate the same text. This is normally not an issue, but see the discussion on theorems on page 91.

\crefname{*type*}{*singular*}{*plural*}	\Crefname{*type*}{*singular*}{*plural*}

These two commands define for a given *type* the text to typeset when a single reference is made and when several references are made by `\cref` and `\Cref`, respectively. For convenience, various types inherit their defaults from other types; e.g., if you change the `section` type, then `subsection` and the other lower levels inherit the new text as well, unless you provide an explicit declaration for them too.

If you define for a given *type* only a `\crefname`, then a corresponding `\Crefname` is automatically provided by uppercasing the first letter in the second and third arguments. Similarly, if only `\Crefname` is provided, then `\crefname` is constructed by the package by applying `\MakeLowercase`.

\creflabelformat{*type*}{*format*}

If you want the labels of a certain *type* formatted in a special way, you can denote that with a `\creflabelformat` declaration. The *format* can be any LaTeX code,[1] and within it `#1` denotes the place where the label (e.g., `\thesection`) is placed, and `#2` and `#3` denote the start and end points of the clickable area if a hyperlink is produced. For example, to add a closing parenthesis to references to an `enumerate` environment, you could write

```
\creflabelformat{enumi}{#2#1)#3}
```

or to remove the parentheses around equation references the solution is to write

```
\creflabelformat{equation}{#2#1#3}
```

\crefrangeconjunction	\crefpairconjunction
\crefmiddleconjunction	\creflastconjunction

To alter the conjunctions between multiple labels, a number of commands exist that contain the material to be inserted; all are changed using `\renewcommand`. Between a consecutive range of labels `\crefrangeconjunction` is added, between pairs `\crefpairconjunction` is used, and for longer lists `\crefmiddleconjunction` and `\creflastconjunction` are added in the appropriate places.

For instance, if you did not like the fact that figures are abbreviated as “figs.” in Example 2-4-8 on page 87 and you prefer a range dash instead of the word “to”, then this can be easily arranged as follows:

```
\usepackage{cleveref}
\crefname{figure}{figure}{figures}
\newcommand\crefrangeconjunction{--}
In \cref{fig:a,tab:a,fig:b,fig:c} we show all
relevant data from the different experiments \ldots
```

2-4-13

In figures 1–3 and table 1 we show all relevant data from the different experiments . . .

[1] Use `\protect` with fragile commands.

`\crefalias{`*type*`}{`*existing-type*`}` `\label[`*type*`]{`*key*`}`

Instead of setting up a (new) *type* with `\crefname`, etc., you can alternatively specify that reference of that *type* should be formatted according to some *existing-type*. This can be useful in some circumstances if you want several counters (types) to use the same referencing format.

You can also use `\label` with an optional *type* argument to overwrite the default type for references to a particular label. For example, if you want to refer to some questions as “assumptions”, the following will do the trick:

```
\usepackage{cleveref}
\crefname{assume}{assumption}{assumptions}
\creflabelformat{assume}{#2(#1)#3}
\begin{equation} a=b\label[assume]{eq}\end{equation}
Starting from \cref{eq} we get \ldots
```

$$a = b \tag{1}$$

2-4-14 Starting from assumption (1) we get …

There are several other adjustments possible with further configuration commands supporting special cases as needed by some languages. Thus, if the above is not sufficient for your needs, consult the package documentation for additional customization possibilities.

Support for multiple languages

Customizing several languages in parallel

So far we covered customizing commands for the main language of a document. If your document uses several languages and you want to customize more than one of them, then you have to get your changes into the language switching mechanism of babel or polyglossia. Here is an example for babel:

```
\usepackage[ngerman,english]{babel,cleveref}
\crefname{figure}{figure}{figures}
\newcommand\crefrangeconjunction{--}
\AtBeginDocument{\addto\extrasngerman{%
    \crefname{figure}{Abbildung}{Abbildungen}%
    \renewcommand\crefrangeconjunction{--}}}
In \cref{fig:a,tab:a,fig:b,fig:c} we have \ldots
\par \selectlanguage{ngerman}
In \cref{fig:a,tab:a,fig:b,fig:c} haben wir \ldots
```

In figures 1–3 and table 1 we have …

In Abbildungen 1–3 und Tabelle 1 haben wir …

2-4-15

Note that the additions to `\extrasngerman` have to be made after the beginning of the document or inside `\AtBeginDocument` to take effect and that we have to use `\renewcommand`, not `\newcommand`, at this point.

Handling theorem-like environments

If you define a new theorem-like environment with the help of `\newtheorem`, then cleveref does not use the counter name as the *type* but instead the environment name that has been set up.

It also automatically assumes that it can use the environment title as the reference text (converted to lowercase if necessary), but it does not make any attempt to set up a plural form as that is too irregular even in English. Thus, if we process the following example, we see that `\Cref` and `\cref` with a single *key* work out of the box, but the last one using a *key-list* fails without further declarations.

Theorem 1 *A theorem.*

Lemma 1 *A lemma.*

Lemma 2 *Another one.*

Lemma 2 is used to prove theorem 1.
But **??** 1**??** 2 need formatting help.

```
\usepackage{cleveref}
\newtheorem{thm}{Theorem}
\newtheorem{lem}{Lemma}
\begin{thm} A theorem.  \label{thm:a}\end{thm}
\begin{lem} A lemma.    \label{lem:a}\end{lem}
\begin{lem} Another one.\label{lem:b}\end{lem}
\Cref{lem:b} is used to prove \cref{thm:a}.    \\
But \cref{lem:a,lem:b} need formatting help.
```

2-4-16

Beside the question marks in the printout we also get warnings like

```
LaTeX Warning: cref reference format for label type 'lem'
               undefined on input line 31.
```

in that case. The remedy is to provide appropriate `\crefname` or `\Crefname` declarations. However, even that is not enough if you set up the theorem-like environments to share a single counter: in that case we suddenly get texts always referring to theorems and not to lemmas where appropriate.

Theorem 1 *A theorem.*

Lemma 2 *A lemma.*

Lemma 3 *Another one.*

Theorem 3 is used to prove theorem 1.
But theorems 2 and 3 need formatting help.

```
\usepackage{cleveref}
\crefname{thm}{theorem}{theorems}
\crefname{lem}{lemma}{lemmas}
\newtheorem{thm}{Theorem} \newtheorem{lem}[thm]{Lemma}
\begin{thm} A theorem.  \label{thm:a}\end{thm}
\begin{lem} A lemma.    \label{lem:a}\end{lem}
\begin{lem} Another one.\label{lem:b}\end{lem}
\Cref{lem:b} is used to prove \cref{thm:a}.    \\
But \cref{lem:a,lem:b} need formatting help.
```

2-4-17

Fortunately, cleveref has a solution for this case too. All you need to do is to use either the amsthm, ntheorem, or thmtools package for theorem-like environments (which is anyway preferable), and then everything comes out correctly.

Lemma 3 is used to prove theorem 1.
Now lemmas 2 and 3 are typeset correctly.

```
\usepackage{amsthm,cleveref}
% Otherwise same setup as in previous example ...
\Cref{lem:b} is used to prove \cref{thm:a}.  \\
Now \cref{lem:a,lem:b} are typeset correctly.
```

2-4-18

Other special considerations

LaTeX's eqnarray *is not supported*

The cleveref package cannot be used together with LaTeX's eqnarray, or, more precisely, you cannot use `\cref` to refer to a `\label` inside such an environment. If you really need this, use `\ref` instead and supply the necessary textual material (e.g.,

"eqs.") manually. In most circumstances it is better to use the environments provided by amsmath anyway, because they offer much better spacing of the equations.

2.4.3 nameref — Non-numerical references

In some documents it is required to reference sections by displaying their title texts instead of their numbers, either because there is no number to refer to or because the house style asks for it. This functionality is provided by the `\nameref` command, available through the nameref package by Sebastian Rahtz (1955–2016) et al. This package is also automatically loaded by hyperref.

For numbered sections and floats with captions, the titles are those that would be displayed in the contents lists (regardless of whether such a list is actually printed). That is, if a short title is provided via the optional argument of a sectioning command or caption, then this title is printed by `\nameref`. This can be somewhat surprising for the reader if the short title of a heading is noticeably different in wording to the title in the body of the document. In contrast, unnumbered sections take their title reference from the printed title. If you use `\nameref` with a label key unrelated to a title (e.g., a label in a footnote, or an enumeration item), it simply displays the title of the surrounding section.

As `\nameref` does not produce the heading number but only its title, you have to additionally use `\ref` if you want to typeset both. More commonly you may want to display the title together with a page reference for which you can use the abbreviation `\Nameref`. Note that this command surrounds the title with single quotes, which may not be to your taste and may lead to strange results if you use other type of quotes elsewhere as we did in the next example.

```
\usepackage{nameref}
\setcounter{secnumdepth}{1}
\section{Textual References}\label{num}
Section \Nameref{num} proves that
it is possible to reference unnumbered sections
by referencing section ‘‘\nameref{unnum}’’.

\subsection[Example]{A Small Example}\label{unnum}
The current section is referenced in
section~\ref{num}.
```

4 Textual References

Section 'Textual References' on page 6 proves that it is possible to reference unnumbered sections by referencing section "Example".

A Small Example

2-4-19 The current section is referenced in section 4.

If hyperref is used, then you can also use `\nameref*`, which works like `\nameref` but prevents a hyperlink to the section. If you load only nameref, both commands have the same effect.

2.4.4 showkeys, refcheck — Displaying & checking reference keys

When writing a larger document, many people print intermediate drafts. In such drafts it would be helpful if the positions of `\label` commands as well as their keys could

be made visible. This becomes possible with the showkeys package written by David Carlisle or the refcheck package by Oleg V. Motygin.

When the showkeys package is loaded, the commands \label, \ref, \pageref, \cite, and \bibitem are modified in a way that the used key is printed. The \label and \bibitem commands normally cause the key to appear in a box in the margin, while the commands referencing a key print it in small type above the formatted reference (possibly overprinting some text). The package tries hard to position the keys in such a way that the rest of the document's formatting is kept unchanged. There is, however, no guarantee for this, and it is best to remove or disable the showkeys package before attempting final formatting of the document.

sec
unused

1 An example

1.1 A subsection

Section 1 shows the use of the showkeys package with a reference to equation (1).

sec
eq

$$a = b \quad (1) \qquad a < b \quad (2) \qquad a > b \quad (3)$$

eq
eq2

```
\usepackage{amsmath,showkeys}
\section{An example}\label{sec}
\subsection{A subsection}\label{unused}
Section~\ref{sec} shows the use of the
\texttt{showkeys} package with a
reference to equation~(\ref{eq}).
\begin{align} a &= b \label{eq}  \\
              a &< b \label{eq2} \\
              a &> b     \end{align}
```

2-4-20

The package supports the fleqn option of the standard classes and works together with the packages of the $\mathcal{A}_{\mathcal{M}}\mathcal{S}$-LaTeX collection, varioref, natbib, and many other packages. Nevertheless, it is nearly impossible to ensure its safe working with all packages that hook into the reference mechanisms.

If you want to see only the keys on the \label command in the margin, you can suppress the others by using the package option notref (which disables the redefinition of \ref, \pageref, and related commands) or the option notcite (which does the same for \cite and its cousins from the natbib package). Alternatively, you might want to use the option color to make the labels less obstructive.

Also supported are the options draft (default) and final. While the latter is useless when used on the package level, because you can achieve the same result by not specifying the showkeys package, draft comes in handy if final is specified as a global option on the class and you nevertheless want to visualize the keys.

If you look at the keys used in Example 2-4-20, then both "unused" and "eq2" are never used in references, and the third equation has an equation number without a label. While the latter is directly visible because there is no boxed key in the margin, the unused keys cannot be identified easily if at all. Nevertheless, all three cases are likely to be either mistakes or leftovers; e.g., some references were intended but never made or misspelled.

To find such problems you can use the package refcheck instead of showkeys. With that package unused labels are shown in the margins surrounded by question marks and in the case of equation tags also underlined. Equations with tags that are not referenced show {?} in the margin. What is not shown are key usage by \ref,

\pageref, or \cite. Thus, by redoing our example with this package, we get the following result:

1 An example

⟨sec⟩ **1.1 A subsection**

?⟨unused⟩? Section 1 shows the use of the refcheck package with a reference to equation (1).

$$a = b \qquad (1)\ \boxed{\texttt{eq}}$$
$$a < b \qquad (2)\ \texttt{?eq2?}$$
$$a > b \qquad (3)\ \{\texttt{?}\}$$

2-4-21

```
\usepackage{amsmath,refcheck}
\section{An example}\label{sec}
\subsection{A subsection}\label{unused}
Section~\ref{sec} shows the use of the
\texttt{refcheck} package with a
reference to equation~(\ref{eq}).
\begin{align} a &= b \label{eq}  \\
              a &< b \label{eq2} \\
              a &> b     \end{align}
```

The checking is also done for \bibitems so that you can easily see if you have any citations in your bibliography that are never referenced in your paper.

If you use the xr package to provide references across different documents, then those can also be verified; for details see the package documentation.

2.4.5 xr — References to external documents

David Carlisle, building on the earlier work of Jean-Pierre Drucbert (1947–2009), developed a package called xr, which implements a system for external references.

If, for instance, a document needs to refer to sections of another document — say, other.tex — then you can specify the xr package in the main file and give the command \externaldocument{other} in the preamble. Then you can use \ref and \pageref to refer to anything that has been defined with a \label command in either other.tex or your main document. You may declare any number of such external documents.

If any of the external documents or the main document uses the same \label key, then a conflict occurs, because the key is multiply defined. To overcome this problem, \externaldocument takes an optional argument in which you can declare a *prefix*. For example, with \externaldocument[A-]{other} all references from the file other.tex are prefixed by A-. So, for instance, if a section in the file other.tex had a \label{intro}, then it could be referenced with \ref{A-intro}. The *prefix* can be any string chosen to ensure that all the keys imported from external files are unique.

Note, however, that if one of the packages you are using declares certain active characters (e.g., : in French or " in German), then these characters should not be used inside \label commands and thus not as part of the *prefix* either.

Citations to external bibliographies

As of 2019 the package also supports referencing \bibitems; i.e., you can cite a bibliography entry with \cite or any of its cousins even if the bibliography is stored in a separate document.[1]

The package does not work together with the hyperref package because both modify the internal reference mechanism. Instead, you can use the xr-hyper package, which is a reimplementation tailored to work with hyperref.

[1] This was originally available as a separate xcite package, written by Enrico Gregorio.

2.4.6 hyperref — Active references

The hyperref package has a long history with many contributors going back to the early days of LaTeX 2ε. The original development was done by Sebastian Rahtz (1955–2016; see page vii), with contributions by Heiko Oberdiek and David Carlisle; later Heiko took over and rewrote and extended the package — so today's comprehensive version is largely due to his efforts. Now the hyperref package is maintained by the LaTeX Project Team.

The package makes it possible to automatically turn all cross-references (citations, table of contents, and so on) into hypertext links. It also supports hyperlinks to external resources, and in addition it offers access to many PDF features, such as bookmarks, etc. The package is described in detail in [57, pp.35–67] and comes with its own extensive manual [167]. In this section we therefore discuss only the most important features useful for day-to-day work, but keep in mind that there is much more available (just in terms of option keys you find more than 100 in the manual).

As has been mentioned in Section 2.1.1 there is a major shift under way during which LaTeX is being modernized to support accessible PDF; adapting hyperref is an important step of the task, and if you start your document with `\DocumentMetadata` to indicate that you want to use the new functionality, a large part of its internal code is different. The LaTeX Project Team works on moving core parts of hyperref directly into the LaTeX kernel, on cleaning up the code, and on extending and standardizing its features.

`\DocumentMetadata` *required!*

These changes have also some impact on the user commands: a few features depend on the new code. Options and commands that are not available or that behave differently without `\DocumentMetadata` are therefore marked with a danger symbol in the following sections.

Using hyperref can be quite easy. Just including it in your list of loaded packages (preferably as the *last* package[1]) suffices to turn all cross-references in your document into hypertext links. For documents viewed on a computer screen, this gives invaluable help for navigating through them.

Configuration possibilities

You may however consider some of the package's default settings not particularly pleasing (such as placing colored boxes around link areas), so many people call the package with a few keys adjusted to taste. The package uses a key/value approach, and most keys can be set when loading the package or later using a `\hypersetup` declaration.[2]

Manually and automatically provided links

Hyperlinks within a document consist of two parts: a region (of text — typically) that, if clicked, instructs the viewing software to jump to a different part in the document (the so called anchor point). This is realized by putting "named" anchors into the target

[1] The hyperref documentation contains a lengthy section discussing deviations to this rule, i.e., in which order certain packages should be loaded in relation to the hyperref package.

[2] Some keys need to be set when the package is loaded because they implement global settings that cannot be altered once set. Even `\hypersetup` is then impossible, except when used in `hyperref.cfg`, the configuration file for the package.

places, surrounding regions that should react to clicks with appropriate commands that invoke some sort of “go to the anchor with a certain name” action.

To be able to jump to the right place, each anchor needs a unique name, and the clickable regions need to know to which “name” they should point. This can be done manually with the following two commands:

`\hypertarget{`*name*`}{`*text*`}` `\hyperlink{`*name*`}{`*clickable text*`}`

The `\hypertarget` typesets the *text* and additionally places an anchor with the name *name* before it. In a different part of the document you can then make a link to it using `\hyperlink`. The clickable region is the *clickable text* argument, and by default this gets surrounded by a box with thin colored borders. If used in the manual way, it is your responsibility to make sure that *name* is unique across the whole document.

In many cases there is no need to produce internal document links manually, because hyperref does this automatically for us behind the scenes. Whenever a command or environment is set up to allow cross-references, hyperref adds an anchor point, and when you use `\ref` or `\pageref`, it surrounds the generated number with a `\hyperlink` so that clicking that number takes you to the section, caption, bibliography item, or whatever else is referenced. In the same way, it adds hyperlinks to the titles (and/or page numbers) in the table of contents, list of figures etc.

If you want to make a reference without a hyperlink, use `\ref*` or `\pageref*` instead. Making hyperlinks to existing `\labels` in the document is also available through the following command:

`\hyperref[`*label*`]{`*text*`}`

This command is useful if you do not want to typeset a normal reference, through `\ref{`*label*`}`, but instead want to refer in *text* to the object the `\label{`*label*`}` is pointing to. Using `\hyperref` turns this *text* into a clickable area. If *text* should additionally contain a `\ref` to display the reference number, use `\ref*` instead to avoid nested links (which do not work).

`\MakeLinkTarget{}`

`\ref`, `\pageref`, and `\hyperref` do not jump to the place where the `\label{`*label*`}` is written but to the last structure before the `\label` that set an anchor. This can have the surprising (at least for `\pageref`) effect that it jumps to a different page than the one shown in the output if, for example, the last section was on a previous page. In such cases an explicit target before the label can be inserted with `\MakeLinkTarget`. This creates the needed target anchor for a correct link if hyperref is loaded.[1]

Links from the table of contents and similar lists

The hyperref package also generates links from the lists generated by the commands `\tableofcontents`, `\listoffigures`, etc., back to the pages with the headings, figures, tables, and so forth. By default, the clickable areas are the heading titles or the captions. This can be changed with the key `linktoc`, which accepts the

[1] The legacy hyperref name for this command is `\phantomsection`, but it is only available if the package is loaded, while `\MakeLinkTarget{}` can be used with and without hyperref.

values `none`, `section` (the default), `page`, or `all` (in which case both the title and page number become hyperlinks).

Links to footnotes

By default, links from footnote markers in the paragraphs to the footnote text at the bottom of the page are automatically added except inside some environments (like `tabularx`) or when packages such as bigfoot are loaded that introduce their own footnote handling. You also do not get a link if you use `\footnotemark` with an optional argument. You can explicitly suppress such links by setting `hyperfootnotes` to `false` if you prefer not to have any footnote links at all.

Links from the bibliography to citations

It is also possible to automatically generate references from the bibliography back to the pages where the bibliography items are cited. This can be helpful, especially during document preparation. This is achieved with the package option `pagebackref` (displaying the page numbers on which a bibliography item is cited) or `backref`. The latter supports the values `section` (displaying the numbers of the sectional units in which the citations are made, the default), `slides` for use in presentations, `page` (same as `pagebackref`), or `none` to prevent them.

There is one important restriction to be aware of: the mechanism to add the links requires that after each `\bibitem` entry there is always an empty line or a `\par` command. If this is missing and the `\bibitem`s directly follow each other, then the links are attached to the wrong place.

The hyperref options for such back references are not relevant when the biblatex package is used to produce the bibliography, because this package implements full support for back references with links directly, and their behavior can and should be adapted by using the relevant biblatex options.

Links from the index entries

Links back from an index to the pages that are referenced are also automatically generated. This can be controlled with the package option `hyperindex`, which can be set to `false` if this is not wanted.

Ensuring unique anchor names

The names for anchors are built by hyperref with the name of the counter and a special representation of the counter called `\theH`⟨*ctr*⟩, which by default expands to `\the`⟨*ctr*⟩. If this representation is not unique across the document and you get warnings about duplicated destination names, you should redefine it, for example, by adding another counter value.

This is a common problem with appendices: their definitions often reset the chapter or section counter to zero and switch the numbering style. We therefore repeat the low-level definition from Section 7 on page 53 to demonstrate what is needed to make it compatible with hyperref. We start with a redefinition of `\appendix`. Here we add a redefinition of `\theH`⟨*ctr*⟩ to get a unique anchor name. We could simply mirror the `\thesection` definition, but in languages different from English `\Alph` is perhaps not usable as an anchor name, so we use a prefix instead. To avoid errors if hyperref is not loaded, we provide also a default definition of `\theHsection`:

```
\providecommand\theHsection{\arabic{section}}
\makeatletter
\renewcommand\appendix{%
   \renewcommand\section{%                % Redefinition of \section...
      \clearpage\thispagestyle{plain}%    %  new page, folio bottom
```

```
      \suppressfloats[t]\@afterindentfalse  %  no top floats, no indent
      \secdef\Appendix\sAppendix}%          %  call \Appendix or \sAppendix
   \setcounter{section}{0}\renewcommand\thesection{\Alph{section}}%
   \renewcommand\theHsection{appendix-\arabic{section}}% for hyperref
 }
\makeatother
```

No change is needed in the `\Appendix` command, but we should tell hyperref the bookmark level:

```
\makeatletter    \providecommand\toclevel@appendix{1}    \makeatother
```

As a minimum, the `\sAppendix` command (implementing the starred form) needs a `\MakeLinkTarget` so that it creates an anchor usable for page references:

```
\newcommand\sAppendix[1]{%                  % Simplified (starred) form
   {\raggedleft\large\bfseries\MakeLinkTarget{}%
    \appendixname\par \centering#1\par}%
    \@afterheading\addvspace{\baselineskip}}
\makeatother
```

The special case of `enumerate` counters, which are typically never unique in a document, is handled internally by hyperref. Another problem can arise from page numbers: hyperref creates for every page an anchor and assumes that every page has a unique name. This is normally the case because roman and arabic page numbers count as different, but it can fail if documents reset the page number after a cover page. The easiest workaround is to set the page number to a negative value for cover pages or to use a different numbering style. If the class hardwires such duplicate page numbers, then another option is to surround the cover pages with the `NoHyper` environment: it disables all hyperref features and so suppresses also the anchor creation.

Links to external resources

It is also possible with hyperref to link to external resources, e.g., to some Internet Uniform Resource Locator (URL) or to a local file, etc. In a PDF file, such links come in three "flavors": links to a URL, links that launch ("run") an external application to view a local file, and links to other PDF files that can be loaded by the PDF viewer. The link types are marked automatically with different colors or link borders that can be specified in `\hypersetup`.

The basic command for such links is `\href`, which attempts to identify the flavor of the link based on some patterns, e.g., if there is a colon in the target or if the file name ends with `.pdf`. For most standard cases this works quite well.

There are now also the more specialized commands `\hrefurl`, `\hrefrun`, and `\hrefpdf` that create the link type as specified by their name and offer some additional options to manipulate the link target. The latter are available only if the command `\DocumentMetadata` has been used at the start of the document.

`\href[`*options*`]{`*link target*`}{`*text*`}`	`\hrefurl[`*options*`]{`*url*`}{`*text*`}`
`\hrefrun[`*options*`]{`*file*`}{`*text*`}`	`\hrefpdf[`*options*`]{`*file*`}{`*text*`}`

The *text* argument is typeset and becomes the clickable area. You can use any kind of formatting within *text* argument — it is just typeset by LaTeX as usual. The first mandatory argument should describe where the link should take us. This can be a website (starting with `https://` or `http://`), but there are many other URL schemes that are defined and supported by many PDF readers. For example,

```
\href{mailto:frank.mittelbach@latex-project.org?subject=Typo found
        in TLC3}{Report Typo in TLC3}
```

would typeset the text "Report Typo in TLC3" in the document and, if clicked, would open the reader's mailing program with my e-mail address and a default subject line prefilled — try it out if you find a typo.[1]

The special characters `#`, `%`, and `~` can be used verbatim in the argument for the link target.[2] This is helpful, because they appear quite often in URLs to web pages.

Non–ASCII links

If the URL contains — as now happens quite commonly — non–ASCII characters, they must be converted into the "percent-encoded" form in the first argument of `\href`; this means, e.g., that a link to the town Köln should be entered as

```
\href{https://www.k%C3%B6ln.de}{Köln}
```

or if used in an argument as

```
\href{https://www.k\%C3\%B6ln.de}{Köln}
```

`\DocumentMetadata` *required!*

For this purpose the `\hrefurl` command offers the option `urlencode`, which does the percent-encoding for you. This makes the LaTeX input considerably longer, but if you need it often, you can also make it the default by setting `href/urlencode` in `\hypersetup`.

```
\hrefurl[urlencode]{https://www.köln.de}{some text}
```

Preset a protocol

Most URLs use the HTTPS protocol. To save some typing, it is possible to preset this protocol with `href/protocol` in `\hypersetup` for `\hrefurl` and `\url`:

`\DocumentMetadata` *required!*

```
\hypersetup{href/protocol=https://}
\hrefurl{www.latex-project.org.de}{some text}
\url{www.latex-project.org.de}
```

Opening files by launching an action

To link to files on the computer you can simply enter the file name in the `\href` argument, or you can launch an action using hyperref's special "`run:`" notation. The first approach works well for PDF files, while a launch action is normally the better choice for all other file types. It instructs the operating system to open the file, and for this

[1] Of course, to work, the viewing software would need to understand the URL schema `mailto:`, and the security configuration would need to allow the browser (or whatever is used for display) to open other applications.

[2] You need to escape them only if `\href` is used inside an argument of another command, e.g., as part of a `\section` title.

to work, the operating system needs to know how to do this.[1] Thus, if you can double-click a *file* in your directory browser and that starts a program to view or process the file, then `run:`*file* in the *url* argument does exactly the same. For example, writing

```
\href{run:resources/video.mp4}{see the video}
```

typesets "see the video", and if clicked it opens — after a security dialog — the file `video.mp4` in the subdirectory `resources` relative to the current document in your default `.mp4` viewer. The optional *options* argument of `\href` can be used if PDF files are opened in this way in Adobe Acrobat viewers. It accepts a number of different keys, e.g., to specify at which `page` the PDF file should be opened; see the package manual for details.

Being able to start other programs in this way out of your document can be very handy, for example, in presentations where you can add little buttons that start an audio or a video presentation, etc.

Links to external PDF files can jump to anchors in these files. If the files have been created with LaTeX and hyperref, the names of the anchors can often be guessed from the representation: in many cases the name is built from the counter name, a period, and the value. For example, anchors to headings are by default constructed as ⟨*heading*⟩.⟨*heading-number*⟩. Thus, to jump to section 1.3 in `manual.pdf`, you can write something like

```
\href{manual.pdf#section.1.3}{see section 1.3 in the manual}
```

Anchor names for other types of numbered objects are less easy to guess. Equations, for example, are often numbered on a per chapter or section basis in document classes. To have a fighting chance for unique names, hyperref constructs such names as `equation.`⟨*section-number*⟩.⟨*equation-counter-value*⟩ (where the section number includes the chapter number if the class has chapters). If in doubt you can take a look into the `.aux` file and look for lines containing the command `\newlabel`.

`\hypersetup{...,` `baseurl` = *baseurl* `,...}`

For URL links it is possible to shorten the *url* arguments by providing a base URL through the key `baseurl`, e.g.,

```
\hypersetup{baseurl=https://www.latex-project.org/}
\href{publications.html}{Publications of the \LaTeX{} Project Team}
\href{help/books.html}  {Books about \LaTeX}
```

This saves a bit of typing and may make later changes easier if most or all URLs have the same base, but be aware that not every viewer program can deal with the fact that the URLs are split into a base part and a remainder and that the base URL is prepended only if hyperref identifies the URL as referring to an external website (e.g., through the `.html` extension). If the viewer thinks it is a local file, no base URL is prepended.

[1] Usually the file extension is associated with a default program to open it, and that is then called.

Establishing a base URL can be done only once for the whole document because this information is written into the PDF catalog. Links to all other places then need to be specified with their complete URL.

`\url{`*url*`}` `\nolinkurl{`*url*`}`

A very common requirement when typesetting an external URL is to suppress normal hyphenation and to allow it to break after slashes and other places. This is provided by the `\url` command from the url package discussed in Section 3.4.7 on page 198. This package is loaded by hyperref and its command is augmented so that you can click the *url* to open it. To typeset a URL without a link, use `\nolinkurl`.

`\DocumentMetadata` *required!*

`\url` has an optional argument and can, like `\hrefurl`, percent-encode its argument if it contains non-ASCII letters. However, this can be used only with Unicode engines — while the link is encoded correctly in pdfTeX, the typeset output in the document is mangled and shows something weird, such as `www.kÃűln.de` or similar, depending on the current font encoding.

Highlighting links

The various links generated either automatically or through the above commands can be highlighted in different ways through a number of keys, either as package options or in a `\hypersetup` declaration. By default, clickable areas are surrounded by a box with thin rules (in color). By specifying one of the following boolean keys, you can change that behavior everywhere in your document.[1]

`colorlinks` Color the text in the clickable area and set the width of the thin rules to zero to make them invisible.

`hidelinks` Do not mark links in any way.

Setting colors

The hyperref package offers a number of keys to change the colors of the text and the borders if they are activated. All keys setting colors accept two color specifications: the name of a color model together with a list of comma-separated numbers, or the extended color syntax such as known from the xcolor package.

```
\hypersetup{ linkcolor = [rgb]{1,0,0} } % red in rgb
\hypersetup{ urlcolor  = red!30!blue }  % mix of red and blue
```

`\DocumentMetadata` *required!*

The color support is built using code from the L3 programming layer (which is part of the format) and is thus available without needing an external color package. Documents not using the new code should load xcolor.

The colors used for the individual links (when using `colorlinks`) can be altered at any time using `\hypersetup` and the following key:

`linkcolor` Color for internal document links.

`filecolor` Color for URLs that open local files.

[1] In older hyperref versions they can be used only in the preamble.

`runcolor` Color for `run:` links.

`urlcolor` Color for externally linked URLs.

`menucolor` Color for "named" links. These are links to menu functions of the PDF viewer; see page 108.

`allcolors` Sets all link colors to the same value.

If you stay with borders around the links (or want to use them in addition to get a particularly colorful result), the names for the keys are the same as those above with `bordercolor` instead of just `color` at the end of the key name. All border colors can be set with `allbordercolors`.

```
\hypersetup{ linkbordercolor = [rgb]{1,0,0} }
\hypersetup{ urlbordercolor = blue!30 }
\hypersetup{ allbordercolors = yellow }
```

The borders around the link areas (when drawn) are by default very thin so that they often become invisible when rendered in the viewing programs.

With `pdfborder` you can adjust their width or reenable the borders if they have been disabled with the `colorlinks` key. This key has a somewhat obscure syntax: you need to supply three numbers: the first two typically zero and the third positive specifying the rule size in pixels.[1]

There also exists the key `pdfborderstyle` that allows you to underline links or place dash boxes around them. The feature is, however, supported only in a few viewers; see the manual for details and examples.

Borders and border styles can also be set for individual link types by using keys such as `urlborder` or `runborderstyle`.


`\DocumentMetadata` *required!*

The hyperref package predefines a number of color schemes for the link colors based on suggestions by users. By default it uses the color scheme `phelype` (named after its author, a member of the LaTeX Project Team). The default colors used by previous versions of hyperref were not to the liking of everyone, but if wanted, they can be restored by using the scheme `primary-colors`.

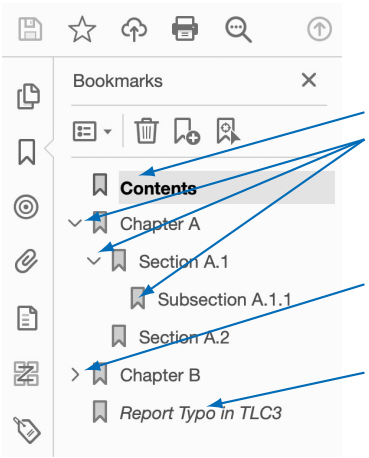

`\DocumentMetadata` *required!*

```
\hypersetup{ colorscheme = primary-colors }
```

Bookmarks a.k.a. outline view

It is possible for a PDF document to contain an outline view, in a manner similar to a table of contents, that can be used for navigating the document in the viewer. A screenshot of such a view in Adobe Acrobat Pro, with some of the formatting options described in this section, is shown in Figure 2.3 on the next page. These "bookmarks" can be (and by default are) automatically produced by the hyperref package. The package option `bookmarks` (default `true`) chooses whether bookmarks are produced at all.

[1] The first two values are used to specify rounded corners, but only a few viewers support this. Even the third value is not uniformly handled, unfortunately, but `0` always omits the border, and a positive value shows it.




```
\DocumentMetadata{}
\documentclass{book}  \usepackage{hyperref}
\hypersetup{next-anchor=toc}  \tableofcontents
\bookmark[dest=toc,bold]{\contentsname}
\bookmarksetup{open,openlevel=3}
\chapter{Chapter A}
\section{Section A.1}
\subsection{Subsection A.1.1}
\section{Section A.2}
\bookmarksetup{openlevel=0}
\chapter{Chapter B}           \section{Section B.1}
\subsection{Subsection B.1.1} \section{Section B.2}
\bookmark[uri =
  {mailto:frank.mittelbach@latex-project.org?subject=Typo
                                              found in TLC3},
  italic,level=0]{Report Typo in TLC3}
```

Figure 2.3: The outline view of a PDF

In older versions this required at least two passes by LaTeX. In the first pass a file with the extension `.out` was written that contained information about each sectional unit plus some bookkeeping data. In subsequent runs the information from the previous run was then placed into the PDF document. Heiko Oberdiek improved this in a separate bookmark package that provided much more sophisticated bookmark management allowing for additional formatting and the use of colors in the bookmarks and that avoided the need of the second compilation.

`\DocumentMetadata` *required!*

The bookmark package has now been merged into hyperref and replaces its legacy code. In older systems or when not using `\DocumentMetadata`, the package bookmark should be loaded either after or instead of hyperref.

`\bookmarksetup{`*options*`}`

Bookmarks have their own command to set up various aspects like the level or the depth.[1] The full list of keys can be found in the documentation [159]; we present here only a few important ones.

Keys that influence how bookmarks are presented

Typically the bookmarks mirror the content of the table of contents and the nesting depth to which bookmarks are added is the value of the counter `tocdepth`. This can be explicitly set and changed through the option `depth`. The key accepts as values integers representing the level but also names for the level like `section`. It can be set anywhere and so allows changing the depth locally. By using a negative value, bookmarks of all levels can be suppressed.

```
\section{section}              % shown
\subsection{subsection}        % shown
\bookmarksetup{depth=section}
```

[1] For historical reasons a few options can also be set with `\hypersetup`.

```
\section{section}              % shown
\subsection{subsection}        % hidden
\bookmarksetup{depth=-1}
\section{section}              % hidden
```

With `numbered` you can decide if the bookmark string should include the section numbers: by default it does not.

With the key `open` you can decide if the view should initially show only the top level (which is the default) or if it should show all bookmarks already opened. Additionally, you can use `openlevel` to request that bookmarks only up to a certain level are initially opened. The value is an integer — unlike the `depth` it does not accept a name — and it can be changed in the document and so allows fine-tuning which parts are opened initially.

Finally, with the keys `bold`, `italic`, and `color`, the bookmark can be formatted. The formatting is always applied to the whole bookmark, and only some PDF viewers honor the settings. For example,

```
\bookmark[dest=toc,bold,italic,color={red!50!green}]{\contentsname}
```

Even if bookmarks are produced, you may not want them to be shown automatically when the document is opened. This is controlled through `pdfpagemode`, which is described below.

Textual data in such bookmarks can contain arbitrary Unicode characters, but complicated formulas or similar constructs are not possible. The hyperref package attempts to parse the titles of sectional units and places only allowed strings into the bookmarks, but in some cases the results are less than suboptimal. For example, suppose you have

```
\section{Discussion of $a \leq b$}
```

as a document heading. First of all this results in three warnings of the form

```
Token not allowed in a PDF string (Unicode):
(hyperref)                removing ‘math shift’ on input line 46.
```

because neither the $ (math shift) nor the `\leq` is allowed. Worse, as a consequence, the text of your bookmark becomes “`Discussion of a b`”, which is simply wrong. In such cases you can help the hyperref package by using `\texorpdfstring`.

`\texorpdfstring{`*TEX string*`}{`*PDF string*`}`

The *TEX string* argument is used when doing normal typesetting, while the second argument is used when writing a bookmark. This argument can even contain UTF-8 characters that are unavailable for typesetting when pdfTEX is used and would normally generate an error. Thus, writing

```
\section{Discussion of \texorpdfstring{$a \leq b$}{a <= b}} % or with
\section{Discussion of \texorpdfstring{$a \leq b$}{a ≤ b}}  % U+2264 character
```

or even just using three dots as the *PDF string* would avoid the warnings and give a better bookmark result.

`\bookmark[`*options*`]{`*bookmark text*`}`

Manual bookmarks

Beside automatic generations of bookmarks through sectioning commands, it is also possible to create bookmarks manually to allow for easy navigation to places that are normally not added to the printed table of contents such as the TOC itself. The target of such a bookmark is given with the key `dest`, which needs as a value the name of the anchor it should point to. Such anchors can be created with `\hypertarget`, but for the table of contents you can also override the name of the automatically created anchor with the key `next-anchor` of `\hypersetup`:

```
\hypersetup{next-anchor=toc}
\tableofcontents
\bookmark[dest=toc,level=0]{\contentsname}
```

Bookmarks executing other actions

Bookmarks not only allow you to jump to places in a document, other actions are possible too. Thus, for example,

```
\bookmark[named=Print]{Print this!}
```

creates a bookmark that — if the PDF viewer supports this action — opens the print dialog. Or to repeat the example from the begin of the section,

```
\bookmark[uri =
    {mailto:frank.mittelbach@latex-project.org?subject=Typo found
                                                          in TLC3}]
    {Report Typo in TLC3}
```

would add the text “Report Typo in TLC3” into the bookmarks and, if clicked, would open the reader’s mailing program with my e-mail address in the same way as the link in the document.

Document properties

If you look at the properties of a PDF document, you find information about title, author, subject, keywords. They can be set in the preamble with keys of the same name but prefixed with `pdf`, e.g.,

```
\hypersetup{pdfauthor   = Frank Mittelbach,
  pdftitle    = {The LaTeX Companion, 3rd edition},
  pdfsubject  = Typesetting,
  pdfkeywords = {document structure, layout, design, LaTeX}}
```

Note the use of braces to hide the commas in the title and keyword list from being misinterpreted as key separators. Like with bookmarks, the values have to be textual data, and hyperref removes unsuitable commands. This is the legacy interface offered by hyperref since its first release. It is, however, only a small subset of the metadata that is these days often required for PDF documents to comply with one or the other standard. It will be therefore eventually superseded by keys offered by `\DocumentMetadata`. Once that happens, these `\hypersetup` keys are deprecated but will remain functional to support reuse of older documents.

PDF presentation possibilities (available with some viewers)

If a PDF document is opened in a viewer program such as Acrobat, it may start with different configurations, e.g., in full screen, on a page different than the first, with or without some menus open, etc. Such variant configurations can already be specified in the source document, though as with many aspects of the hyperref package, the actual behavior depends on the viewer used: they all work with Adobe's Acrobat programs but not necessarily elsewhere. The remainder of the section therefore describes the situation with Acrobat software. Some of the keys also work with other viewers, but the results may differ from viewer to viewer, so you need to check.

Perhaps the most important key is `pdfpagemode` with which you can control the initial viewing layout. Possible values are `UseNone`, `UseThumbs`, `UseOutlines` (i.e., show bookmarks), `FullScreen`, `UseOC` (when using overlay layers[1]), and `UseAttachments`.

Normally the document window title shows the file name displayed, but if you prefer to see its title, then add the key `pdfdisplaydoctitle`. The title should be set with `pdftitle` for this; using only the command `\title` is not enough.

By default Acrobat starts out with both a menu bar and a tool bar (or pane) open. Their settings are controlled through the keys `pdfmenubar` and `pdftoolbar`. Especially the latter takes up a lot of space, so I prefer to turn it off by setting its key to `false`.

Pages can be presented either as single pages or two pages side by side, and one can flip them or ask for continuous scrolling. This is controlled through the key `pdfpagelayout` that accepts six different values: `SinglePage` (flip pages when pressing down or up keys), `OneColumn` (single pages with scrolling), `TwoPageRight` (two pages with odd pages on the right), and `TwoColumnRight` (ditto with scrolling).

A simple-minded way to determine recto and verso pages

Note that Acrobat does not look at the logical page number but simply uses the physical one to determine odd and even. It therefore also offers `TwoPageLeft` and `TwoColumnLeft`, but neither helps if your pages are not continuous. In such a case you really have to add empty pages into your document so that it is displayed correctly.

By default the first physical page of PDF file is shown. To specify a different starting page, use the key `pdfstartpage`. Again, if your logical pages are specially numbered, you may have to count to determine the right physical page number to

[1] For example with the help of the `ocgx2` package by Alexander Grahn.

use as a value. With individual links that open PDF documents, you can also specify a starting page with the key `page` in the optional argument to `\href`.

If you open PDF documents through `\href` links, then Acrobat replaces the current document with the new one, which is often not desired. With the global boolean key `pdfnewwindow`, you can specify that a new window should be used instead in all such cases. Alternatively, you can do this on individual links in the optional argument to `\href`.

`\DocumentMetadata` *required!*

It is also possible to add transition options. They typically have an effect only if you view the PDF in full-screen mode and add animations between page switches like pages flying into the screen or dissolving into the background.

```
\hypersetup{pdfpagetransition={style=Glitter,duration=2,
                               direction=180}}
```

Other miscellaneous features

For Adobe Acrobat viewer software the hyperref package offers some special support for accessing the program menus through the command `\Acrobatmenu`. It allows you to define clickable areas that act as if you have selected the corresponding menu. A huge number of menu items are supported (see the package documentation), but probably only a few of them are likely to be useful.

```
\Acrobatmenu{FullScreen}{F} \Acrobatmenu{FitWidth}{W}
\Acrobatmenu{NextPage}{R}   \Acrobatmenu{PrevPage}{L}
```

This places “F”, “W”, “L”, and “R” onto the page, and if you click them, the Acrobat menu action is carried out, e.g., your document changes size or advances to the next page, etc. This can be helpful occasionally, but if you know the corresponding keyboard shortcuts, it does not gain you that much. Also note that the clickable area is only as big as the glyph(s) in the second argument, so if you try to make them inconspicuous, there is not much to click unless you use gray or even white. The border color around the link area can be set with key `menubordercolor` or, if the link text is colored, with `menucolor`.

The package also offers a useful set of commands to build PDF or HTML forms with fields, check boxes, radio buttons, etc. If you are interested in that kind of functionality, consult the package documentation for details.

As mentioned in the beginning, the package offers more than one hundred keys to adjust its behavior in certain situations of which we covered only the most important ones in this section. If you require some feature that appears not to be possible, study the extensive package documentation — it may well exist after all.

2.5 Document source management

In the final section of this chapter we discuss tools that help you archiving your documents as well as reliably exchanging them with others, e.g., journal publishers.

We start with environments that hold the contents of a file, which is then extracted when the document is processed, allowing you to combine several files in one document. We then look at ways to gather information on "used files" for archival purposes. This is followed by looking at two programs that take such information to produce an archive with all relevant files included: bundledoc, which saves only the text and package files used and produces fairly small archives, and mkjobtexmf, which does a more thorough job and also includes fonts and similar binary data. Which of them is more suitable depends on your use case.

Finally, we briefly discuss the latexrelease package, which offers you a way to roll back your LaTeX installation to an earlier date without the need to install a previous release explicitly. There are limits to what it can achieve, but it is a good addition to LaTeX's insurance that your documents can be processed successfully without any changes in the output for long periods of time.

2.5.1 Combining several files

When sending a LaTeX document to another person, you may have to send local or uncommon package files (e.g., your private modifications to some packages) along with the source. In such cases it is often helpful if you can put all the information required to process the document into a single file.

`\begin{filecontents}[`*option-list*`]{`*file name*`} ... \end{filecontents}`

For this purpose, LaTeX provides the environment `filecontents`. This environment takes one mandatory argument, the name of a file[1]; its body consists of the contents of this file. The `\begin` and `\end` tags should be placed on lines of their own in the source. In particular, there should be no material following them, or you will get LaTeX errors.

If LaTeX encounters such an environment, it tries to find the mentioned file name. If it cannot, it writes the body of the environment verbatim into a file in the current directory and inform you about this action. Conversely, if a file with the given name was found by LaTeX, it informs you that it has ignored this instance of the `filecontents` environment because the file is already present on the file system.

The *option-list* argument allows you modify this behavior. If you specify `nosearch`, then only the current directory is searched for the file, not the whole TeX tree. This is useful if you want to write, for example, a local version of a configuration file, such as `graphics.cfg`, which would otherwise not appear in your local directory. If you specify `force` (or `overwrite`), then the file is always written, even if it already exists in the current directory or somewhere in the TeX installation tree. Use this option with caution because you can clobber files this way by mistake.[2] You can

[1] If no extension is specified, the actual external file name is the one LaTeX would read if you used this name as an argument to `\input`, i.e., adding the extension `.tex`.

[2] The environment refuses to write to `\jobname.tex` — disaster is assured if you overwrite your own input file. However, other files might be equally important!

silence any warnings from the `force` key by also specifying `nowarn`, in which case warnings are only written to the `.log` file.

By default the generated file gets a few comment lines (using % as a comment character) added to the top to announce that this file was written by a `filecontents` environment:

```
%% LaTeX2e file 'foo.txt'
%% generated by the 'filecontents' environment
%% from source 'test' on 2022/04/22.
```

If this is not appropriate — for example, if the file is not a LaTeX file — use the option `noheader` in which case these extra lines are not produced. Alternatively, you can use the `filecontents*` environment instead, which is just a short way to set this option.

In older LaTeX formats the content of such a file was restricted to ASCII characters — with other characters all bet were off. These days essentially any Unicode character should be admissible.

If you use `filecontents` to ship all files necessary to process your document in a single master file, then it is best to place the environment(s) at the very top of the file so that they are written out before they are needed when processing the document.

There are, however, also use cases where one would want to write files somewhere inside the document body. For example, if you have some material that is reused several times, you could write it to a file and then load that file via `\input` wherever necessary. Other use cases are packages that require their input in external files (ltxtable is an example). In that case you can keep your data where it belongs in your source and write it to a file prior to using it. If you are using `filecontents` for such purposes, it is best to add the `force` option, because otherwise you are likely to be puzzled by the fact that you change your data and nothing happens in your document (because the file was already written out in a previous run).

2.5.2 Document archival information

For archival purposes or sharing or collaborating on documents, it is often important to record (and usually collect) all files needed for processing a document. This needs their correct versions to faithfully re-create the document at a later stage or in a different place. For this a number of tools and programs are available.

As a simple solution LaTeX already offers the command `\listfiles`, which records all files that are opened with `\documentclass`, `\usepackage`, `\include`, `\input`,[1] `\includegraphics`, etc. Suppose you process the following document

```
\documentclass[12pt]{article}  \usepackage{lmodern}
\listfiles
\begin{document} Hello, world! \end{document}
```

[1] Files opened with `\input` are recorded only if you use the recommended syntax with a braced argument. The primitive plain TeX syntax that delimits the file name with spaces is not supported!

then as a result your transcript file will show the following list of files, possibly with different version numbers if your installation is older or younger:

```
 *File List*
 article.cls    2021/02/12 v1.4n Standard LaTeX document class
  size12.clo    2021/02/12 v1.4n Standard LaTeX file (size option)
 lmodern.sty    2015/05/01 v1.6.1 Latin Modern Fonts
  ot1lmr.fd    2015/05/01 v1.6.1 Font defs for Latin Modern
l3backend-pdftex.def    2021-05-07 L3 backend support: PDF output (pdfTeX)
***********
```

As you can see, this shows the document class, the class option file, the package used, and one font definition file for Latin Modern, but it is clearly missing everything else related to font usage. Thus, if the fonts used in your document do not exist elsewhere (or in a different version), then the results of processing your document may differ without a way to determine the cause.

Nevertheless, it goes a long way towards resolving issues when collaborating with others or experiencing a problem that others do not seem to have: good advice in such cases is to add `\listfiles` to the document and compare the results on different installations. In many cases this already pinpoints the reason for different behavior.

2.5.3 snapshot, bundledoc — Document archival and verification

The `snapshot` package by Michael Downes (1958–2003) uses the same approach as `\listfiles` for collecting file information about a document but presents it in a way that it can be automatically verified at a later stage or on a different installation. This is particularly useful when collaborating or when one want to archive documents and record this information as part of the document itself.

To enable it, you have to place the package in the first line of your document using `\RequirePackage[error]{snapshot}` even before the `\documentclass`. Without any further options to the package, this will then write a file with the extension `.dep` (for dependencies) containing the following lines if applied to our example document:

```
\RequireVersions{
  *{application}{pdfTeX}  {0000/00/00 v1.40.22}
  *{format} {LaTeX2e}     {2021-06-01 v2.e}
  *{package}{snapshot}    {2020/06/17 v2.14}
  *{class}  {article}     {2021/02/12 v1.4n}
  *{file}   {size12.clo}  {2021/02/12 v1.4n}
  *{package}{lmodern}     {2015/05/01 v1.6.1}
  *{file}   {ot1lmr.fd}   {2015/05/01 v1.6.1}
  *{file}   {l3backend-pdftex.def}{2021-05-07 v3}
}
```

Up to this point this is not much difference than when using `\listfiles`, except that the information is placed into a separate file and slightly more structured. However,

as a next step you can copy the content of this file into your document directly after loading the package, which makes it the information of record for this document. From now on this data is checked at each run, and if any differences are found, they raise a warning or an error.

For example, assume that you collaborate with some people on writing the "Hello World" short story and their TeX installation has an obsolete Latin Modern package somewhere in their `texmf` tree, then they will see the following error message

```
! Package snapshot Error:
  Required version 2009/10/30 v1.6 of lmodern.sty and
  provided version 2008/12/01 v1.5 do not match.
```

if they attempt to run the document. If you prefer to generate just warnings instead of errors, use the option `warning` (or no option). It is also possible to restrict file information verification just to dates, versions, or major version numbers by using one of the options `date`, `version`, or `major-version` — if the latter is applied in our example, there would be no error because the major version is 1 for both files.

The `.dep` file produced by `snapshot` can also be used to produce an archive with all or some of the files it lists, by using the `bundledoc` program by Scott Pakin. This is particularly useful if you want to send your document with all the necessary files to somebody else and do not want to worry about missing anything relevant. For example, journals often request all source files in addition to the camera-ready PDF. In that case running

```
bundledoc --verbose --localonly --include=⟨myfile⟩.pdf ⟨myfile⟩.dep
```

does the trick, and you get an archive file[1] containing the final PDF and everything that is required and not part of the main TeX installation. Of course, it requires an up-to-date `.dep` file; i.e., you have to include the `snapshot` package as described above and process your document with it.

Alternatively, or in addition, you can use `--exclude=`*string* to exclude all files whose names contain that string, and with `--include`, you can explicitly request additional files otherwise not included to be added to the archive like we did above. For example, you may want to include your bibliography databases (and not just the resulting `.bbl` files used by the document), which can be achieved with `--include="*.bib"`. Both options can be used as often as necessary. The `--verbose` option, as used above, gives some progress information.

Without any of the options above, `bundledoc` includes all files listed in the `.dep` file, which is more suitable for archival purposes. But do not forget that some files important for the final results (such as font files) are not included. The advantage is that the archive is noticeably smaller in size compared to those produced by

[1] The exact type of archive depends on your operating system; on Windows it is typically a `.zip`, on Unix or macOS a `.tar.gz` file. The exact behavior can be controlled through configuration files.

mkjobtexmf discussed in the next section. Usually a workable approach is to additionally archive the yearly TEX Live distributions as fonts change less often. However, for 100% accurate results it might be required to archive all files for a given project using mkjobtexmf.

By default, bundledoc flattens the directory structure and places all files in the archive next to each other. With `--keepdirs` the original structure is preserved.

With the option `--config` you can select a configuration file, for example, `--config=miktex.cfg` for `.zip` archives on MiKTEX. Another interesting one is `texlive-unix-arlatex.cfg`, which generates a single LATEX file including all other files through `filecontents` environments. How to define your own configuration file is described in the documentation where you also find details on a few other options that may be useful in some cases.

You can omit the extensions `.dep` and `.cfg` for the dependency and the config file, so on MiKTEX, for example, we could write

```
bundledoc --config=miktex --localonly --include=⟨myfile⟩.pdf ⟨myfile⟩
```

to prepare a `.zip` file for a journal submission.

2.5.4 mkjobtexmf — Providing a minimal TEX file tree

To find out exactly which files are used by a TEX engine when processing a document, most modern engines offer the command-line option `-recorder`. If it is used, then a file with the extension `.fls` is produced that contains information on all files that the engine opened for reading or writing, one per line. With our "Hello World" example this amounts to 28 lines, and after removing the duplicates (LATEX opens most files twice for reading), the following 16 remain, among them various configuration and font files and the format file:

```
INPUT /usr/local/texlive/2023/texmf-dist/fonts/enc/dvips/lm/lm-rm.enc
INPUT /usr/local/texlive/2023/texmf-dist/fonts/map/fontname/texfonts.map
INPUT /usr/local/texlive/2023/texmf-dist/fonts/tfm/public/cm/cmr12.tfm
INPUT /usr/local/texlive/2023/texmf-dist/fonts/tfm/public/lm/rm-lmr12.tfm
INPUT /usr/local/texlive/2023/texmf-dist/fonts/type1/public/lm/lmr12.pfb
INPUT /usr/local/texlive/2023/texmf-dist/tex/latex/base/article.cls
INPUT /usr/local/texlive/2023/texmf-dist/tex/latex/base/size12.clo
INPUT /usr/local/texlive/2023/texmf-dist/tex/latex/lm/lmodern.sty
INPUT /usr/local/texlive/2023/texmf-dist/tex/latex/lm/ot1lmr.fd
INPUT /usr/local/texlive/2023/texmf-dist/tex/latex/snapshot/snapshot.sty
INPUT /usr/local/texlive/2023/texmf-dist/web2c/texmf.cnf
INPUT /usr/local/texlive/2023/texmf-var/fonts/map/pdftex/updmap/pdftex.map
INPUT /usr/local/texlive/2023/texmf-var/web2c/pdftex/pdflatex.fmt
INPUT /usr/local/texlive/2023/texmf.cnf
INPUT myfile.aux
INPUT myfile.tex
```

For 100% accuracy, all of them, except the `.aux` file, should be archived, and doing this with the `--include` option of bundledoc would be rather cumbersome. This is

where the mkjobtexmf program by Heiko Oberdiek comes into play. If you execute

```
mkjobtexmf --verbose --copy --jobname myfile   # without extension
```

then LaTeX is run (using the `-recorder` option) on the file `myfile.tex`. The resulting `.fls` file is examined, and all files in the above listing are then copied into the directory `myfile.mjt` using a standard setup of `texmf` subdirectories. For archival, all that remains is to zip up this directory and store it in a safe place. Note that the `--copy` or `--force-copy` is essential for this to work: without it mkjobtexmf adds links to the files, not physical copies.[1] The `--copy` does not overwrite existing files in the target `texmf`, whereas `--force-copy` does. The latter is useful because it means that updates are properly accounted for if you run the program repeatedly and want to make sure that the latest versions are inside the tree.[2]

Always mandatory is the option `--jobname` to specify the file to run and the default destination directory. The `--verbose` displays information about what mkjobtexmf does and is sometimes helpful.

If you prefer a flat structure with all files directly in the `myfile.mjt` directory, specify the option `--flat`. If your document should be processed with one of the Unicode engines, you can specify this too, e.g., by using `--cmd-tex lualatex` for LuaTeX.

There are a number of further options to tweak the program behavior including defining the destination directory (`--destdir`), the LaTeX file name to process if it has an extension different from `.tex` (`--texname`), and several others. `--help` produces a concise but useful reference.

Existing files are never changed in the destination directory

If you use mkjobtexmf for archival purposes as described above, then you should be aware of one important aspect in the program behavior. It always only adds *new* material to its destination directory but never deletes from it nor does it replace any existing link to a file with a copy of the file or vice versa. If the purpose is to speed up processing, that is fine, but for archiving the final result, it might mean that the archive contains files no longer used or contains links where it should contain copies because you forgot to specify `--copy` on the first invocation. Even worse, it may not contain the latest version of your source files if they have changed since the first time the program was used.

2.5.5 The rollback concept for LaTeX and individual packages

Keeping your LaTeX installation up-to-date and using the latest packages is usually a good approach because that means you get the latest corrections and feature updates. The LaTeX universe is well-known for its unparalleled backward compatibility:

[1] A `texmf` tree containing only links does not take up much space, but speeds up the processing of a document because it contains only the files necessary for the document to run.

[2] I used this during the production of this book to store all packages used in the book in a source control system (with history). That enabled me to keep track of changes that happened to the packages while writing the book.

reprocessing documents decades old with a modern LaTeX is normally not a problem, and very seldom requires adjustments by the user.

However, there are cases where packages change their interfaces in incompatible ways or where you have worked around a problem and now that the problem is solved, your workaround no longer works.

For situations like this, LaTeX introduced in 2015 a rollback concept for the LaTeX kernel as well as for document classes and packages, allowing the LaTeX maintainers to make corrections to the software while continuing to maintain backward compatibility to the highest degree. With its help you can explicitly ask LaTeX to revert its code to a version that was current on a specific date, and the software tries its best to undo changes to match this state.

To request a kernel rollback to its state at a given *date*, you use the latexrelease package by the LaTeX Project Team. For example,

```
\RequirePackage[2016-01-01]{latexrelease}
```

would result in undoing all kernel modifications (corrections or extensions) released between January 1, 2016, and the current date.[1] Undoing means reinstalling the definitions current at the requested date and normally also removing new commands from TeX's memory so that `\newcommand` and similar declarations do not fall over because a name is already declared.

This mechanism helps in correctly processing older documents that contain workarounds for issues with an older kernel and issues that have since been fixed in a way that would make the old document fail, or produce different output, when processed with the newer, fixed kernel.

If necessary, the latexrelease package also allows for rolling the kernel forward without installing a new format. For example, if your current installation is dated 2016-04-01 but you have a document that requires a kernel with date 2018-01-01, then this can be achieved by starting it with

```
\RequirePackage[latest]{latexrelease}
```

provided you have a version of the latexrelease package that knows about the kernel changes between the date of your kernel and the requested date. Getting this version of the package is simple as the latest version can always be downloaded from the Comprehensive TeX Archive Network (CTAN). Thus, you are able to process your document correctly, even when updating your complete installation is not advisable or is impossible for one or another reason.[2]

[1] There are a few exceptions because some modifications are kept: for example, the ability to accept date strings in ISO format (i.e., 2016-01-01) in addition to the older LaTeX convention (i.e., 2016/01/01). These are not rolled back because removing such a feature would result in unnecessary failures.

[2] For example, this might help when you work on Overleaf (an online LaTeX portal). At the time of writing this book, Overleaf was about a year behind the current LaTeX release. Of course, in that case you may also have to upload individual packages into your account, if they have specific new features that you want to use.

Typical scenarios

A typical example, for which such a rollback functionality would have provided a major benefit (and will do for packages in the future), is the caption package by Axel Sommerfeldt. This package started out under the name of caption with a certain user interface. Over time it became clear that there were some deficiencies in the user interface; to rectify these without making older documents fail, Axel introduced caption2. At a later point the syntax of that package itself was superseded, resulting in caption3, and then that got renamed back to caption. So now older documents using caption will fail, while documents from the intermediate period require caption2 (which is listed as superseded on CTAN but is still distributed in the major distributions). So users accustomed to copying their document preamble from one document to the next are probably still continuing to use it without noticing that they are in fact using a version with defective and limited interfaces.

Another example would be the fixltx2e package that for many years contained fixes to the LaTeX kernel. In 2015 these were integrated into the kernel so that today this package is an empty shell, only telling the user that it is no longer needed. However, if you process an old document (from before 2015) using rollback, and that document loads fixltx2e, then of course fixes originally provided by this package (like the corrections to the floats algorithm) would get lost as they are now neither in the kernel nor in the "empty" fixltx2e package if that does not roll back as well — fortunately it does, so in reality it is not quite an empty shell.

A somewhat different example would be the amsmath package, which for nearly a decade did not see any corrections even though several problems have been found in it over the years. If such bugs finally get corrected, then that would affect many of the documents written since 2000, since their authors may have manually worked around one or another deficiency of the code. Of course, as with the caption package, one could introduce an amsmath2, amsmath3, ... package, but that puts the burden on the user to always select the latest version (instead of automatically using the latest version unless an earlier one is really needed).

The document-level interface

By default LaTeX automatically uses the current version of any class or package — and prior to offering the new rollback concept it always did that unless the package or class had its own scheme for providing versioning, either using alternative names or using hand-coded options that select a version.

Global rollback

With the new rollback concept all the user has to do (if they want a document processed with a specific version of the kernel and packages) is to add the latexrelease package at the beginning of the document and specify a desired date as the package option, e.g.,

```
\RequirePackage[2018-01-01]{latexrelease}
```

This rolls back the kernel to its state on that day (as described earlier), and for each package and the document class, it checks if there are alternate releases available and

selects the most appropriate release of that package or class in relation to the given date.

Individual rollback

There is further fine-grain adjustment possible: both `\documentclass` as well as `\usepackage` have a second (less known) optional argument that up to now was used to allow the specification of a "minimal date". For example, by declaring

```
\usepackage[colaction]{multicol}[2018-01-01]
```

you specify that multicol is expected to be no older than the beginning of 2018. If only an older version is found, then processing such a document results in a warning message:

```
LaTeX Warning: You have requested, on input line 12, version
   '2018-01-01' of package  multicol, but only version
   '2017/04/11 v1.8q multicolumn formatting (FMi)' is available.
```

The idea behind this approach is that packages seldom change syntax in an incompatible way, but more often add new features: with such a declaration you can indicate that you need a version that provides certain new features.

The new rollback concept now extends the use of this optional argument by letting you additionally supply a target date for the rollback. This is done by prefixing a date string with an equal sign. For example,

```
\usepackage{multicol}[=2017-06-01]
```

would request a release of multicol that corresponds to its version in June 2017.

So assuming that at some point in the future there will be a major rewrite of this package that changes the way columns are balanced, the above would request a fallback to what right now is the current version from 2017-04-11. The old use of this optional argument is still available because existence or absence of the = determines how the date is interpreted.

The same mechanism is available for document classes via the `\documentclass` declaration and for `\RequirePackage` if that is ever needed.

Preparing your document for posterity

Specifying a rollback date is most appropriate if you want to ensure that the behavior of the processing engine (i.e., the kernel and all packages) corresponds to that specific date. In fact, once you are finished with editing a document, you can preserve it for posterity by adding this line at the top of your document:

```
\RequirePackage[today's-date]{latexrelease}
```

This would mean that it is processed a little more slowly (because the kernel may get rolled back and each package gets checked for alternate versions), but it would have the advantage that processing it a long time in the future will probably still work without the need to add that line later.

Specifying a version instead of a date

However, in a case such as the caption package or, say, the longtable package, that might eventually see a major new release after several years, it would be nice to allow the specification of a "named" release instead of a date: for example, a user might want to explicitly use version 4 rather than 5 of longtable when these versions have incompatible syntax or produce different results.

This is also now possible if the developer declares "named" releases for a package or class: one can then request a named version simply by using this second optional argument with the "name" prefixed by an equal sign. For example, if there is a new version of longtable and the old (now current) version is labeled "v4", then all that is necessary to select that old version is

```
\usepackage{longtable}[=v4]
```

Note that there is no need to know that the new version is dated 2018-04-01 (nor to request a date before that) to get the old version back.

The version "name" is an arbitrary string at the discretion of the package author — but note that it must not resemble a date specification; i.e., it must not contain hyphens or slashes, because these confuse the parsing routine.[1]

Erroneous input may have strange effects

The user interface is fairly simple, and to keep the processing speed high, the syntax checking is therefore rather light and rather unforgiving if it finds unexpected data. Basically any string containing a hyphen or a slash triggers the date parsing, which then expects two hyphens (in case of an ISO date) or two slashes (otherwise) and other than these separators, only digits. If it does find anything else, chances are that you get a "`Missing \begin{document}`" error or, perhaps even more puzzling, a strange selection being made. For example, `2011/02` may mean to you February 2011, but for the parsing routine it is some day in the year 20 A.D. That is, it gets converted to the single number `201102` so that when this number is compared numerically to, say, `20000101`, it is the smaller number, i.e., earlier, even though the latter is the numerical representation of January 1, 2000. Bottom line: do not misspell your dates, and all is fine.

The package writer interface

The commands to set up the rollback functionality in packages and classes are described in Appendix A.6.1 on page →II 693; for more details on the concepts, see [137].

[1] Of course, more sophisticated parsing could fix this, but we opted for a fast and simple parsing that scans for slashes or hyphens with no further analysis.

CHAPTER 3

Basic Formatting Tools — Paragraph Level

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.

In this chapter we look at such tools, starting at the micro level; larger structures are covered in Chapter 4. The first section covers different aspects of paragraph formatting, 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 similar topics. This is followed by a look at handling special characters such as ellipses, dashes, underscores, or spaces.

In the third section we discuss generated or specially formatted text, i.e., counter values represented as ordinals or cardinals, fractions formatted for use in running text, and in particular the acro package for consistently managing acronyms and abbreviations. A special focus is given to scientific notation provided by the siunitx package, which forms the last and rather lengthy topic of this section.

The fourth section then covers various way of highlighting and quoting text. This includes a number of generally useful packages as well as some more specialized ones that are occasionally useful.

Section 3.5 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. In the final section we take a quick look at different helper packages for document development, e.g., how to add different kind of notes, copy-editing marks, or change bars to your documents.

3.1 Shaping your paragraphs

Paragraph justification in TEX and 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 are 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.4.3 discusses how to trace the algorithm. If you are interested in delving further into the matter of automatic paragraph breaking, refer to *The TEXbook* [84, chap. 14], which describes the algorithm in great detail, or to the very interesting article by Michael Plass and Donald Knuth on the subject, which is reprinted in *Digital Typography* [99].

Downside of global optimization

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 an important project (like the third 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\CElinebreak{\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.

Interword spacing

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 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 also considers poorer but hopefully 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, because that allows TEX to produce a single arbitrarily bad line (like this one) to keep the rest of the paragraph perfect. 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 adds 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. It uses a `\tolerance` of 9999 together with an `\emergencystretch` of 3 em.

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

[1] The optimum is font defined; see Section 9.8.1 on page 745.

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 (1949–2011) [183].

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 because 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, because there is nothing to counteract it, TEX's paragraph-breaking algorithm normally chooses line breaks that avoid hyphenated words.

"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}
```

3-1-1

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 `ragged2e` in the next section for alternatives). Nevertheless, it can be useful to place individual objects, like a graphic, flush left to the margin, especially because 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`.

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 6.2.2).

It is also important to realize that the command forms always apply to whole paragraphs, even if used in the middle of a paragraph. TEX uses the setting active at the *end* of a paragraph to decide how to justify the text. This means that if using, for example, `\centering` inside a group, you have to ensure that the paragraph ends within that group, otherwise your request is ignored or partially ignored.


End of paragraphs matter!

3.1.1 ragged2e — Improving unjustified text

Above we discussed the deficiencies of LATEX's `flushleft` and `flushright` environments if used for normal text. The package `ragged2e`, written by Martin Schröder and now maintained by Marei Peischl, 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 distributes excess space equally to all spaces. As a result, the interword spaces 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.

The available parameters and their default values are shown in Table 3.1 on the following page. They are used as values for `\parindent`, `\leftskip`, `\rightskip`, and `\parfillskip`, whenever one of the corresponding `ragged2e` commands or

The default values

[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.

Parameter	Default	Parameter	Default
\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	1em	\JustifyingParfillskip	0pt plus 1fil

Table 3.1: Parameters used by ragged2e

environments is called. Using em values in the defaults (see Table 3.1) means that special care is needed when loading the package, because 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 3.1, one can instruct the package to initially mimic the original LaTeX settings by using the option originalparameters and then changing the parameter values as desired afterwards.

Unjustified setting as the default

To set a whole document unjustified, you can specify document as an option to the ragged2e package. For the purpose of justifying individual paragraphs in such a document the package offers the command \justifying and the environment justify. 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 3-1-1 on page 122):

```
\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.''
```

"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."

3-1-2

Unjustified settings in narrow columns

In places with narrow measures (e.g., \marginpars, \parboxes, 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.

Spurious underfull box warnings

There is, however, one problem that you should be aware of if you use the command `\RaggedLeft` or `\Centering` with very little text (i.e., less than a single line): you may get strange "Underfull box" warnings such as

```
Underfull \hbox (badness 10000) in paragraph at lines 25--25
 []\T1/ptm/m/n/10 ragged left text
Underfull \hbox (badness 5893) in paragraph at lines 26--27
 []\T1/ptm/m/n/10 centered text
```

even though the result looks (and is) correct. For example, the above warnings have been generated during the processing of the next example:

3-1-3

ragged right text

ragged left text

centered text

```
\usepackage{ragged2e}
\RaggedRight ragged right text \par
\RaggedLeft  ragged left text  \par
\Centering   centered text
```

The reason is that with `ragged2e` there is only very limited flexibility in each line compared to `\raggedleft` or `\centering` where the white space on one or both sides can stretch arbitrarily. `\RaggedRight` on the other hand is usually fine, because there we still have a fully stretchable `\parfillskip` at the end of the paragraph.

Overloading the original commands not recommended

Thus, while it is tempting to overload the standard LaTeX definitions with the new commands (using the package option `newcommands`) to avoid the need to typeset the somewhat tedious mixed-case names, it cannot really be recommended. At least `\centering` is very often used to center a single object such as a graphic in a `figure` environment, and each such case would then result in a spurious warning.

3.1.2 nolbreaks — Preventing line breaks in text fragments

To prevent a line break at a space inside a paragraph LaTeX offers ~ denoting an unbreakable space that you can use instead of an ordinary one, e.g., `A.~Einstein` to ensure that the initial and surname are not split apart. If you (additionally) want to ensure that a word is not hyphenated, you can put it into an `\mbox`, e.g., `A.~\mbox{Einstein}`.

However, to keep several words together, it is not a good idea to place them together with the spaces between them into a single `\mbox`, because inside a box a space has always its nominal width and does not react to the justification of the line, which means that you can end up with noticeably uneven spacing.[1] For high quality it is therefore necessary to `\mbox` all words individually and place a ~ between each of them — which is fairly cumbersome. To simplify this task Donald Arseneau has written the small package nolbreaks that offers a single command.

[1] Exemplified in this paragraph by boxing "it is not a good idea" in the first line.

`\nolbreaks*{`*text*`}`

The *text* does not break across lines, but spaces inside still participate in paragraph justification as expected. If you use the starred form, then the line before the unbreakable text is allowed to run short (like ragged-right) as shown below. You can also load the package with the option `ragged` in which case `\nolbreaks` behaves like its starred form.

However, to keep several words together, it is not a good idea to place them together with the spaces between them into a box; use `\nolbreaks` instead.

```
\usepackage{nolbreaks} \sloppy
However, to keep several \nolbreaks*{words together,}
it is not a good idea to place them together
with the spaces between them into \nolbreaks{a box};
use \verb=\nolbreaks= instead.
```

3-1-4

The command does not work in all circumstances, e.g., you cannot have verbatim material in its argument, and spaces hidden inside braces or commands can still create breakpoints, but in most situations it offers a simple and readable method for fine-tuning your text. Note that you may need a higher `\tolerance` or `\sloppy` if you add many unbreakable chunks to your paragraphs.

3.1.3 microtype — Enhancing justified text

As mentioned before, TEX uses an algorithm for line breaking that attempts to globally optimize the paragraphs according to a set of parameters weighing different (often conflicting) goals, such as unevenness in the white space distribution, incompatible lines (with respect to word space size), length of the paragraph, number of consecutive hyphens, etc., against each other.

There are, however, a number of further aspects that improve the paragraph quality not taken into account by the original TEX algorithm. Support for them is due to the work of Hàn Thế Thành who developed pdfTEX, which is now the standard TEX engine,[1] and thus these improvements are available to everybody [62, 63, 65].

Already Donald Knuth discussed the use of "hanging punctuations" as an exercise in the *TEXbook* [84, p. 394f] and gave the following example:

"What is hanging punctuation?" asked Alice, with a puzzled frown. 'Well, y'know, actually,' answered Bill, 'I'd rather demonstrate it than explain it.' "Oh, now I see. Commas, periods, and quotes are allowed to stick out into the margins, if they occur next to a line break." 'Yeah, I guess.' "Really! But why do all your remarks have single quotes, while mine are double?" 'I haven't the foggiest; it's weird. Ask the author of this crazy book.'

3-1-5

Optical alignment a.k.a. protrusion feature

As you can see, all punctuation marks and quotation characters are placed outside the text body into the margin. This is a special version of a general principle of optical alignment: to achieve optimal vertical alignment of the text at the margins,

[1] The features discussed in this section are also (with minor variations) available in the Unicode engines XeTEX and LuaTEX and can thus be used with any modern TEX engine.

it is necessary to take the glyph shapes into account and allow some glyphs to protrude slightly into the margin because otherwise the line would appear to be slightly indented — hanging punctuation is just an extreme variant of this principle. How much to protrude depends on the glyph shape and the amount of whiteness it produces. It thus depends on the font being used and may need adjustments accordingly for optimal results. However, even if you do not have specially tailored values for the fonts used in your document, you achieve noticeable improvements by applying a set of default values based on "typical" glyph forms.

The hz algorithm a.k.a. expansion feature

A second type of improvement introduced with pdfTeX was in the incorporation of the *hz*-algorithm named after its inventor Hermann Zapf (1918–2015). He realized that (certain) letter shapes can be slightly expanded or compressed without being noticeable to the reader and that this extra flexibility in the text can be used to improve justification. For example, instead of just enlarging the word space (possibly beyond acceptable limits), one can slightly widen most letters in a line, thereby achieving a much more consistent gray value of the whole paragraph. In fact, the additional flexibility introduced this way may lead to different set of line breaks that would otherwise not be possible. This can then avoid overfull lines that would be otherwise produced by TeX's algorithm if it could not satisfy all requirements posed by the line-breaking parameter settings.

Both features and configuration possibilities to tailor them are made available through the microtype package[1] by Robert Schlicht. It has been used with excellent results throughout this book and is one of the standard packages the author loads in the preamble of nearly every document.

Tracking, kerning, and spacing feature

Three other micro-typographical features are supported but not activated by default (because applying them does not always lead to improvements). These are automatic letterspacing of Small Capitals and possibly other fonts (discussed in Section 3.4.6), extra kerning around individual characters, and interword spacing adjustments based on the character before the space (kind of an extension to the `\spacefactor` concept of TeX). The features are enabled with the options `tracking`, `kerning`, and `spacing`, respectively. Tracking can be used with pdfTeX and LuaTeX, while the other two features are available only with pdfTeX to date.

Package options

In many cases it is fully sufficient to simply load the package with a `\usepackage` declaration (without any option) in the preamble and let it apply its magic behind the scenes using its default configuration. It then automatically applies character protrusion and (if possible) font expansion.

For the latter the engine has to be either pdfTeX or LuaTeX, you have to use only scalable outline fonts (i.e., no bitmap fonts generated from METAFONT sources[2]), and the engine must be set to generate Portable Document Format (PDF) and not Device Independent File Format (DVI). For details of what is possible in DVI mode, see

[1] We discuss most aspects of the package here; its letterspacing features are covered in Section 3.4.6.

[2] Bitmap fonts usually produce a fatal error; see page →II 735. There are ways to work with them, but the fonts need to be specially tailored for this.

the manual [179]. Saying it differently, the package applies as many micro-typographic improvements as can be expected to work correctly given the circumstances.

Turning them on (or off)

It does, however, offer a number of key/value options to globally adjust the behavior and as we see later, also methods to tailor the behavior depending on fonts, font sets, and the context inside the document. The options `protrusion` and `expansion` can be used to change or turn on or off the respective feature by giving the value `true` (default), `false`, or `compatibility`. The latter restricts the features to act only on single lines after the line-breaking algorithm has acted. This ensures that the line breaking with or without microtype is identical, but at the same time it limits the positive effects that the package can offer and is therefore useful only in special situations.

One can also specify a *font set name* as a value, in which case only fonts belonging to this set use the feature. For details on this advanced usage, consult the microtype manual [179].

Protrusion control

The option `factor` can be used to tailor the protrusion feature, and it expects an integer value between 0 (no protrusion) and 1000 (full protrusion). For example, specifying a value of 500 means that the currently defined protrusion for every character is halved. To showcase the results let's first repeat Example 3-1-5 on page 126 with a `factor` of 0 (which is equivalent to not using protrusion at all):

"What is hanging punctuation?" asked Alice, with a puzzled frown. 'Well, y'know, actually,' answered Bill, 'I'd rather demonstrate it than explain it.' "Oh, now I see. Commas, periods, and quotes are allowed to stick out into the margins, if they occur next to a line break." 'Yeah, I guess.' "Really! But why do all your remarks have single quotes, while mine are double?" 'I haven't the foggiest; it's weird. Ask the author of this crazy book.'

3-1-6

You can nicely observe the different line breaks that we get with just the normal TEX algorithm. Now repeat this but with a `factor` of 500, in which case the punctuation and quote characters stick out somewhat into the margins.

"What is hanging punctuation?" asked Alice, with a puzzled frown. 'Well, y'know, actually,' answered Bill, 'I'd rather demonstrate it than explain it.' "Oh, now I see. Commas, periods, and quotes are allowed to stick out into the margins, if they occur next to a line break." 'Yeah, I guess.' "Really! But why do all your remarks have single quotes, while mine are double?" 'I haven't the foggiest; it's weird. Ask the author of this crazy book.'

3-1-7

The line breaks are now identical to Example 3-1-5 even though we use a smaller amount of protrusion. Note that the protrusion feature does not generate more flexibility for the paragraph breaking (in contrast to the expansion feature). It only alters the line breaks because characters at the margins appear unconditionally smaller than without the feature turned on and thus change the attractiveness of individual breakpoints so that TEX may decide to choose a different set.

Also note that in the examples above protrusion was only specified for the punctuation and quote characters to implement "hanging punctuation". For proper optical alignment, one would have to specify protrusion for many more characters. As a showcase for optical alignment, you can look at any paragraph in this book (except

for the examples) where many characters stick out to a small degree into the margin to give the impression of vertical alignment at both sides.

Expansion control

To control expansion, a few more options are available. The options `stretch` and `shrink` define how much fonts are allowed to be expanded or compressed. They expect an integer value that is multiplied by $\frac{1}{1000}$ of the character width. Thus, to allow a maximum of 1.5% expansion you would specify `stretch=15`. The default value for both is `20`, and if you specify only `stretch`, then its value is also inherited by `shrink`.

As the next example shows, you should be conservative when allowing fonts to stretch or shrink. The idea is to improve the typographic quality by producing paragraphs with more uniform grayness and fewer hyphenated lines by offering the line-breaking algorithm more choices. With most fonts a variation range of ±2% yields reasonable results, but already above 3% you may notice the stems getting bigger or thinner, which can be distracting. Furthermore, some shapes are somewhat distorted when only stretched horizontally, and above a certain point this may become noticeable and thus reduce, rather than enhance, the quality.

Stretching or shrinking text too much is a bad idea!	*-15%*
Stretching or shrinking text too much is a bad idea!	*-5%*
Stretching or shrinking text too much is a bad idea!	*-2%*
Stretching or shrinking text too much is a bad idea!	*(natural)*
Stretching or shrinking text too much is a bad idea!	*2%*
Stretching or shrinking text too much is a bad idea!	*5%*
3-1-8 Stretching or shrinking text too much is a bad idea!	*15%*

To lessen the impact of distorted shapes, microtype offers the option `selected` in which case such shapes are expanded or compressed at a smaller rate than others. This may allow slightly higher overall limits, but on the other hand one has to realize that it also means that characters next to each other may stretch at different rates and thus show different stem widths. In our example, the characters “trix!” are fully expanded, while all others are expanded only by 70%. Whether or not this is then really an improvement is probably a matter of taste.

Stretching or shrinking text too much is a bad idea!	*-3%*
Stretching or shrinking text too much is a bad idea!	*-2%*
Stretching or shrinking text too much is a bad idea!	*(natural)*
Stretching or shrinking text too much is a bad idea!	*2%*
Stretching or shrinking text too much is a bad idea!	*3%*
Stretching or shrinking text too much is a bad idea!	*5%*
3-1-9 Stretching or shrinking text too much is a bad idea!	*15%*

For best results with expansion you need to use scalable fonts (which is fortunately not a problem any longer), and it is advisable to generate PDF output, though neither is an absolute must. If need be, it is possible to use a DVI-based workflow, and

even bitmap fonts can be prepared for the task, but it requires a more complicated setup and specially tailored font support files; see the manual [179] for details.

Miscellaneous options

There are a few other options that can become handy once in a while. Among them is `activate`, which is simply a shorthand for setting both `protrusion` and `expansion` to the same value. Then there is `verbose`, which outputs extra information into the transcript file (useful for debugging) or, if given the value `errors`, stops if it thinks it encounters a questionable situation. Once you have investigated all warnings, you can also give it the value `silent`.

Specifying the option `babel` directs microtype to adjust the typesetting to the language(s) used in the document. This feature exists for only a few selected languages so far.

The package supports the option `disable` that disables *all* processing if given. As a result, processing is much faster when microtype does nothing, but then line and page breaks are likely to change.[1]

By default microtype loads its configuration from the file `microtype.cfg`. You can bypass this by using the option `config` and specify a different configuration file (without the extension `.cfg`). This way you can maintain and use your own configuration setup if you do not like the default values.

All options except for `config` can alternatively be used in the argument to `\microtypesetup` to define or alter the desired setup in the preamble. Furthermore, this command can also be used inside the document body to temporarily disable or enable any of the micro-typography features, e.g.,

```
\microtypesetup{expansion=false}
```

but otherwise changing the features is no longer possible.

Configuring the machinery

As already indicated, the micro-typography features supported by microtype all require tailoring to the individual fonts used to achieve best results. While that sounds no doubt daunting, the good news is that the package already comes equipped with ready-made protrusion settings for more than a dozen common font families, and for most others the default profile is likely to give you good results too. For expansion the default of ±2% is also likely to work universally, and if not, it is easy to change for a document. In summary, the general usage information provided above should be sufficient for most real-life situations.

We therefore give only a few examples for setup commands to enable you to skim through the configuration files and grasp what they are setting up. This may not be enough to embark on the somewhat tedious task of providing a fully hand-tailored setup for a new font family, but if you are one of the few people[2] interested in that, all necessary information can be found in [179].

[1] In older versions of the package this option was called `draft`, which was rather unfortunate, because specifying `draft` on the document class made microtype stop working. You can restore the previous behavior, if you really want to, by specifying `disable=ifdraft`.

[2] Rest assured that Robert would be happy to hear from you that you have worked on the integration of another font family.

`\SetProtrusion[`*key/value list*`]{`*set of fonts*`}{`*protrusion settings*`}`

The `\SetProtrusion` declaration lets you define *protrusion settings* for an arbitrary number of characters for a given *set of fonts*.

The *protrusion settings* consists of *character*`={`*integer-tuple*`}` where the *integer-tuple* defines the protrusion for left and right margin for the *character*. Zero or no value means no protrusion, and `1000` denotes full protrusion. The *character* can be given as a UTF-8 character, a LaTeX command, and in a few other ways. For example

```
A = {50,50},  \AE = {50, }, : = { ,500},  - = {400,500}
```

lets "A" protrude by 5% on both sides and "Æ" by 5% on the left only. Both the colon and hyphen protrude with half of their width into the right margin, and on the left the hyphen also protrudes with 40% of its width. If you specify a base character such as "A", then microtype knows that there are several other characters, e.g., "À, Á, Â, Ã, Ä, Å, Ą, Ă", that should inherit the setting, and it does that automatically[1] for you.

There are many ways to specify the *set of fonts*, but very often it is enough to specify the supported encoding(s) and the font family name (in New Font Selection Scheme (NFSS) convention); for further details, see [179, §4].

In the optional argument you can specify a number of key values: with `name` you can give your setting a name, which is useful when reading the output from the `verbose` option. More importantly, this allows you to refer to such a setting in a later declaration with a `load` key, thereby extending or modifying it for special situations.

For example, the main protrusion settings for Computer Modern fonts are named `cmr-default`; additions specific to `T1` encoding are named `cmr-T1` (loading `cmr-default` first). And because Latin Modern fonts are very similar to Computer Modern, the protrusion declaration for that family is then simply this:

```
\SetProtrusion [ name = lmr-T1, load = cmr-T1 ]
   { encoding = {T1,LY1}, family = lmr }
   { \textquotedblleft = {300,400}, \textquotedblright = {300,400} }
```

That is, it loads `cmr-T1`, which in turn loads `cmr-default` and then makes two changes. All of this applies to any font in the font family `lmr` in either `T1` or `LY1` encoding (but not in `OT1` or other encodings).

If you like the look and feel of hanging punctuation, here is how Example 3-1-5 from page 126 was produced using Latin Modern fonts:

```
\usepackage{lmodern}
\usepackage[expansion=false,verbose]{microtype}
\LoadMicrotypeFile{cmr}
```

[1] Of course, if ever needed, that is customizable too, using `\CharacterInheritance`. With pdfTeX the standard settings are most likely adequate for all fonts. However, with the Unicode engines it is more likely that font-specific adjustments are needed, simply because OpenType fonts have many more characters (or miss some) so that a single default is not sufficient. Details on that is given in [179].

```
\SetProtrusion [ name = lmr-hangingp  ]
    { encoding = {T1,OT1}, family = lmr }
    { . = { ,1000}, {,} = { ,1000}, ; = { ,1000}, : = { ,1000},
      \textquotedblleft = {1000, }, \textquotedblright = { ,1000},
      \textquoteleft    = {1000, }, \textquoteright    = { ,1000}  }
```

This example is interesting on several accounts: we explicitly disabled expansion to reproduce the same line breaks as in the TEXbook. With expansion, TEX would have found a "better" or at least different alternative. The protrusion settings as such hold no surprises: each of the punctuation and quote characters fully protrudes into the respective margin. You may also want to do the same to the hyphen character. This was not done, because again it would result in different line breaks compared to the TEXbook.

Nothing happens when making config changes

The surprising line is the second one — without it nothing happens. The problem is that when microtype encounters a font family for the first time in a document, it attempts to load a configuration file named `mt-`⟨*family*⟩`.cfg` and applies the declarations it contains. If that file also contains a declaration for the *set of fonts* that we try to customize, it overwrites our carefully drafted setup in the preamble of the document. The solution here is to load that configuration file (using `\LoadMicrotypeFile`) before we make our declarations so that our settings overwrite the default and not the other way around.

A slight complication is that some font families share such configuration files. In our case we have to load the one for the `cmr` font family, because that also contains the settings for `lmr`. The basic advice here is to use the `verbose` option if something does not seem to work, because that tells you what microtype loads for which font and what gets overwritten and with that information it is easy to make the right adjustments.

Supporting the development visually

If you develop your own protrusion set for a font family or if you want to verify that the currently used one is sufficient and adequate, you can use the companion package microtype-show that offers a number of check and test commands, such as `\ShowProtrusion` or `\ShowCharacterInheritance`. These commands produce test output by displaying the current settings both numerically as well as visually. This is a great development and debugging facility; for details see the microtype package documentation.

`\SetExpansion[`*key/value list*`]{`*set of fonts*`}{`*expansion settings*`}`

Expansion is normally applied uniformly to all characters, and the necessary settings (if any) are done through the package options `stretch` and `shrink`. If, however, the option `selected` was used, then certain characters expand more than others, and the settings for this can be done through `\SetExpansion`.

The *expansion settings* are a list with elements of the form *character=factor* where *factor* is an integer between 0 (no expansion) and 1000 (full expansion).

The *set of fonts* argument limits the declaration to a group of fonts in the same way as was already discussed for `\SetProtrusion`. For example, the default settings

(defined in `microtype.cfg`) used for all fonts in the major text encodings look as follows:

```
\SetExpansion [ name = default ] { encoding = {OT1,OT4,QX,T1,LY1} }
   { A = 500, a = 700,  \AE = 500, \ae = 700,  B = 700,  b = 700,
     C = 700, c = 700,    D = 500,   d = 700,  E = 700,  e = 700,
     F = 700,             G = 500,   g = 700,  H = 700,  h = 700,
     K = 700, k = 700,    M = 700,   m = 700,  N = 700,  n = 700,
     O = 500, o = 700,  \OE = 500, \oe = 700,  P = 700,  p = 700,
     Q = 500, q = 700,    R = 700,             S = 700,  s = 700,
     U = 700, u = 700,    W = 700,   w = 700,  Z = 700,  z = 700,
     2 = 700, 3 = 700, 6 = 700, 8 = 700, 9 = 700                  }
```

In the *key/value list* argument you can again use `name` and `load`, though with expansion, these keys are less likely to be needed. The keys `stretch` and `shrink` overwrite the global defaults or the values given on the package level. One application of this is when you provide special settings for named contexts as we see below.

Providing context

By default all declarations made with `\SetProtrusion` and friends apply globally throughout the document. There is, however, the possibility of specifying that they should be activated only in a specific context. This is done by using the key `context` in the *key/value list* argument and assigning it a *context* name. In that case the declaration is activated only if we are within that context.

```
\microtypecontext{context spec}
\begin{microtypecontext}{context spec} ... \end{microtypecontext}
\textmicrotypecontext{context spec}{text}
```

You can inform microtype that it is in a specific context in three different ways: with `\microtypecontext` a new context is started that continues to the end of the current scope, or you can use the environment form of that, or you can use the command `\textmicrotypecontext` in which the context applies to the *text* argument. The *context spec* is a comma-separated list of assignments of the form *feature* = *context* where *feature* is `protrusion`, `expansion`, `tracking`, `kerning`, or `spacing` and *context* is the name you assigned in the declaration. To reset the context (if not automatically reverted through scoping) you can provide an empty *context name*.

Specifying tracking, extra kerning, and adjusted spacing

The features described below cover some concepts that are not activated by default but need to be explicitly enabled through options, i.e., `tracking`, `kerning`, and `spacing`, respectively. Furthermore, the kerning and spacing features are available only with pdfTeX, while tracking can also be used with the LuaTeX engine. An interesting article on the background of these features can be found in [64].

```
\SetTracking[key/value list]{set of fonts}{tracking amount}
\SetExtraKerning[key/value list]{set of fonts}{kerning settings}
\SetExtraSpacing[key/value list]{set of fonts}{spacing settings}
```

The \SetTracking declaration allows you to specify extra spaces between all characters of a font or a set of fonts in order to achieve letter spacing. This is discussed further in Section 3.4.6 on page 191.

The \SetExtraKerning declaration provides a way to specify for individual characters some additional space to be added on either side of the glyph. This is useful with languages that have typographical traditions requiring such spacing around some characters (typically punctuations). In French, for example, an extra space is required in front of ":", ";", "?", and "!", and with a declaration such as

```
\SetExtraKerning[name=french-default, context=french, unit=space]
                {encoding = {OT1,T1,LY1}}
                { : = {1000,},  ; = {500, }, ! = {500,},  ? = {500,} }
```

this can be automatically provided without the need to alter the input sources. Note the use of the context key, which limits the declaration to a context named french so that you can restrict its usage as needed.

To communicate with babel and install different contexts per language, microtype offers the declaration \DeclareMicrotypeBabelHook. In the next example we use this to specify special kerning for French text. The \SetExtraKerning declaration is already in that form part of the microtype default settings, so we can make use of it without the need to repeat it in the example.

```
\usepackage[english,french]{babel}
\usepackage[kerning]{microtype}
% \SetExtraKerning ... as above
\DeclareMicrotypeBabelHook{french}{kerning=french}
FRENCH: Je ne parle pas français! \par
\selectlanguage{english} ENGLISH: I speak English!
```

FRENCH : Je ne parle pas français !
ENGLISH: I speak English!

3-1-10

Finally, with the \SetExtraSpacing declaration you get granular control over the width and behavior of interword spaces. For each font TEX maintains three dimensions (called \fontdimens: see page 745) that describe the default width of a space, the amount by which that space can stretch, and the amount it can shrink when doing paragraph justification. In addition, there is a "space factor" table where for each character a factor is specified by which these font dimensions are multiplied when that character is directly in front of a space. This is the reason why spaces after punctuation characters appear larger (when \nonfrenchspacing is in force — the default) and appear uniform (when \frenchspacing is specified).

The problem with TEX's approach of a space factor is that you not only get larger or smaller spaces with the help of these factors; in addition, they are also applied to the stretching and shrinking components and so in spaced-out lines spaces after punctuation characters may become excessively large.

The pdfTeX engine extends the TeX functionality by providing individual controls on a per-character and per-font basis for all three dimensions, and the `\SetExtraSpacing` declaration offers an interface to that. For example,

```
\SetExtraSpacing[unit=space]
                {font={*/*/*/*/*}}{ . = {2000,0,0} }
```

would double the space after a period without altering its stretch and shrink component. The mechanism can also be used to implement optical spacing, that is, slightly altering the standard width of a space depending on the shape of the preceding character. The default configuration files for microtype provide some settings for this. However, this is of course semi-optimal given that only a character to the left of the space can influence its width.

For further details on all three features refer to the package manual [179] and study the default configuration files that provide standard settings from which you may want to start if you make changes.

Disabling selected ligatures

Using ligatures, e.g., “ffi” instead of “ffi”, usually improves the appearance, and the fact that LaTeX can produce them automatically (if available in the font) is normally a good thing. There are occasions where you want to prevent this from happening, e.g., in compound words like “Auflage” instead of “Auflage”, (because it is a special “Lage” not a “Flage”), but these are rare and need to be handled on an individual basis. The babel package offers some support for this; see Example 13-3-7 on page →II 311. However, there are also situations where one wants to disable ligatures (or some ligatures) altogether and for this microtype provides `\DisableLigatures`.

`\DisableLigatures[`*start characters*`]{`*set of fonts*`}`

With the *set of fonts* argument one specifies for which fonts ligatures should be suppressed and with the optional *start characters* argument, it is possible to restrict it to only a subset of ligatures. For example, Latin Modern Typewriter has ligatures for “--” becoming a single “–” and “<<” and “>>” generating “«” and “»”, respectively. To prevent this you could write

```
\DisableLigatures[-,<,>]{family = lmtt}
```

Note that for technical reasons you can specify only the ligature start character, thus by specifying, for example, `f`, you disable all ligatures starting with “f”.

Some special considerations when using microtype

As mentioned before it is usually just enough to load microtype in the preamble without any further adjustments to the document body. There are, however, a few cases to watch out for, and it might pay to temporarily turn off some or all of the package features or guide them in difficult situations.

Being pedantic about protrusion

Protrusion, for example, works automatically in normal paragraphs, but in certain situations TEX does not consider the text whose boundary requires vertical alignment (and thus protrusion) as being at the margin, because there is some intervening material. For instance, the bullet in an `itemize` is seen by LATEX as being part of the line, while for a human reader the text of that item is what needs visual alignment. Thus, without help, the first character on that line would not protrude, while the first character on the next line would.[1]

`\leftprotrusion{`*text*`}`	`\rightprotrusion{`*text*`}`
`\leftprotrusion`	`\noprotrusion`

In case protrusion is not done automatically or not done correctly, you can force or prevent it with these commands. `\leftprotrusion` and `\rightprotrusion` add a protrusion correction to the left or right of *text*, respectively. You can use also `\leftprotrusion` without an argument, in which case it scans the input for text characters, and if it finds one (which may be a ligature), it applies protrusion to this character. The version without argument is slightly less efficient, but it has the advantage that you can use it in places where you do not know what text is following, e.g., at the beginning of a `tabular` cell:

```
\begin{tabular}{l>{\leftprotrusion}p{9cm}r}
```

This is unfortunately not possible for protrusion on the right: with the command `\rightprotrusion` you always have to use the argument.

There is also `\noprotrusion` that prohibits protrusion in all cases. This command is already defined in the LATEX format so you can use it in command definitions whether or not microtype gets loaded in your document.

Interaction with TOC-like lists

If protrusion is turned on while a table of contents is being typeset, then the page numbers at the right might protrude into the margin. That in turn makes the left-hand side of the page number column somewhat uneven, which may be noticeable if the numbers start with the same digits. The solution in such a case is to use `\microtypesetup` to set `protrusion` to `false`, i.e.,

```
\microtypesetup{protrusion=false}
\tableofcontents \listoftables \listoffigures
\microtypesetup{protrusion=true}
```

Standard LATEX avoids this problem, but with older document classes or special definitions for such lists it may still happen.

Interaction with `verbatim`

Similarly, both protrusion and expansion are not really wanted if you typeset code material verbatim. After all, the idea of using a mono-spaced font in `verbatim` environments is to ensure that the characters appear perfectly aligned above each other, and either feature may spoil that alignment. Thus, setting both features (or `activate`

[1] There is work under way to help LATEX to recognize this particular case automatically, so in the future the `itemize` case may work automatically. Currently, the microtype package attempts to patch `\item`, but this does not always work and may have to be prevented with the `nopatch` option.

as a shorthand) to `false` in front of such environments solves this problem. However, you may not want to litter your source with such declarations, especially if you have many such environments.[1]

A possibly better alternative is to incorporate the setting into the environment itself using `\AddToHook` with a reasonably new LaTeX as follows:

```
\AddToHook{env/verbatim/begin}{\microtypesetup{activate=false}}
```

There is no need to undo the setting because the environment forms a group so that at its end the change is automatically undone. If you are doing that, then you may want to also update `\tableofcontents` and friends in a similar way:

```
\AddToHook{cmd/tableofcontents/before}{\microtypesetup{protrusion=false}}
\AddToHook{cmd/tableofcontents/after} {\microtypesetup{protrusion=true}}
```

In this case we have to turn microtype on again after the command.

Interaction with ragged2e

Another situation where expansion is often not really helpful is the case of typesetting unjustified text using the ragged2e package, discussed in Section 3.1.1 on page 123. Because this package tries to avoid extreme raggedness by making short lines appear "underfull" to the paragraph-breaking algorithm, microtype mistakenly tries to help by expanding the fonts in such lines. Thus, nearly all lines in unjustified paragraphs get expanded, which is not a desired state of affairs.

On the other hand, microtype is still useful if there is a need to slightly shrink a line to make it fit. Instead of completely disabling expansion when typesetting unjustified text, it is better to define a special context in which only shrinking but not stretching is allowed. This can be done as follows (and with the help of `\AddToHook` directly attached to `\RaggedRight` and similar commands):

```
\SetExpansion[ context = ragged, stretch = 0, shrink = 30 ]
              { encoding = {OT1,T1,TS1} } {}
\AddToHook{cmd/RaggedRight/before}{\microtypecontext{expansion=ragged}}
```

There is usually no need to restore the context in this case, because that happens automatically when the scope of `\RaggedRight` ends.

3.1.4 parskip — Adjusting the look and feel of paragraphs

In the majority of publications, paragraphs are typeset justified with the first line indented by some fixed amount and the last line run short based on the line-breaking results of the paragraph material. Because of the indentation at the left in the first line and the white space on the right in the last line, readers can easily identify the paragraphs without any further visual signals. Therefore, there is typically no extra vertical space added between paragraphs, which saves space. This type of layout is used by most LaTeX document classes and is what you see on most pages in this book.

[1] Verbatim-like environments done with listings or fancyvrb are not affected because due to their implementation, protrusion or expansion is automatically prevented.