Joaquín Ataz-López

A not so short introduction

to ConTEXt Mark IV

A not so short introduction to ConTEXt Mark IV
Version 1.6 [January 2, 2021]

Original title: Una introducción (no demasiado breve) a ConTEXt Mark IV
English Translation: A good friend who wishes to remain anonymous.

Version history:

- August 18, 2020: Version 1.0 (Spanish only): Original document.
- August 23, 2020: Version 1.1 (Spanish only): Correction of minor errors, typos and misunderstandings by the author.
- September 3, 2020: Version 1.15 (Spanish only): More errors, typos and misunderstandings.
- September 5, 2020: Version 1.16 (Spanish only): More errors, typos and misunderstandings as well as some very minor changes to make the text clearer (I hope).
- September 6, 2020: Version 1.17 (Spanish only): The number of minor errors I am finding is incredible. I would just need to stop re-reading the document to find no more!
- October 21, 2020: Version 1.5 (Spanish only): Introduction of suggestions and correction of errors reported by NTG-context users.
- January 2, 2021: Version 1.6: Corrections suggested after a new and careful reading of the document, on the occasion of its translation to English. This is the first version in English.

Table of Contents

Preface*

Gentle reader, this is a document about ConTEXt, a typesetting system derived from TEX, which, in turn, is another typesetting system created between 1977 and 1982 by Donald E. Knuth at Stanford University.

ConTEXt was designed for the creation of documents of very high typographical quality – either paper documents or documents designed to be displayed on the screen of a computing device. It is not a word processor or text editor, but, as I said before, a *system*, or in other words a suite of tools aimed at typesetting documents, understood as the graphic layout and visualisation of the different elements of the document on the page or on screen. ConTEXt, in summary, aims to provide all the tools needed to give documents the best possible appearance. The idea is to be able to generate documents that, besides being well written, are also "beautiful". In this respect, we can mention here what Donald E. Knuth wrote when presenting TEX (the system on which ConTEXt is based):

> *If you merely want to produce a passably good document—something acceptable and basically readable but not really beautiful—a simpler system will usually suffice. With TEX the goal is to produce the* finest *quality; this requires more attention to detail, but you will not find it much harder to go the extra distance, and you'll be able to take special pride in the finished product.*

When we prepare a manuscript with ConTEXt, we indicate exactly how this must be transformed into pages (or screens) whose typographical quality is comparable to what can be obtained with the best of the world's print shops. To do this, once we have learned the system, we need little more work than what is needed to normally type up the document in any word processor or text editor. In fact, once we have gained a certain ease in handling ConTEXt, our total work is probably less if we bear in mind that the main formatting details of the document are described

* This preface began with the intention of being a translation/adaptation to ConTEXt of the preface to "The TEXBook", the document that explains *everything you need to know about TEX*. Ultimately, I had to deviate from that; however, I have retained some snippets that I hope, for those who know it, will offer some *echoes* of it.

globally in ConTEXt and we are working with text files that are – once we are accustomed to them – a much more natural way of dealing with the creation and editing of documents; other than the fact that these kinds of files are much lighter and easier to deal with than the heavy binary files belonging to word processors.

There is a considerable amount of documentation on ConTEXt, almost all of it in English. What we might consider to be the *official* distribution of ConTEXt – called "ConTEXt Standalone"[1] – for example, contains some 180 PDF files of documentation (the majority of it in English, but some in Dutch and German) including manuals, examples and technical articles; and on the Pragma ADE web (the company that gave birth to ConTEXt) there are (on the day I did the count in May 2020) 224 downloadable documents, most of which are distributed with the "ConTEXt Standalone" but some others as well. Just the same, this huge documentation is not particularly useful for learning ConTEXt since, in general, these documents are not aimed at a reader who knows nothing about the system but wants to learn it. Of the 56 PDF files that "ConTEXt Standalone" calls "manuals", there is only one that assumes that the reader knows nothing about ConTEXt. This is a document entitled "ConTEXt Mark IV, an Excursion". This document, however, as its name indicates, limits itself to presenting the system and explaining how to do certain things that can be done with ConTEXt. It would be a good introduction if it were followed up by a somewhat more structured and systematic reference manual. This manual does not exist and the gap between the document relating to the ConTEXt "Excursion" and the rest of the documentation is too great.

In 2001, a reference manual was written and can be found on the Pragma ADE web site; but despite this title, one the one hand it was not designed to be a *complete manual*, while on the other it was (is) a text aimed at the previous version of ConTEXt (called Mark II) and is therefore quite out of date.

In 2013, the manual was partially updated but many of its sections were not rewritten and it contains information relating to both ConTEXt Mark II and ConTEXt Mark IV (the current version), without always making it totally clear what information refers to each of the versions. Perhaps this is why this manual is not to be found among the documents included in "ConTEXt Standalone". Yet despite these defects, the manual continues to be the best document for beginning to learn ConTEXt once we have read the introductory "ConTEXt Mark IV, an Excursion". Also very useful for starting out in ConTEXt is the information to be found in its wiki which, at the time this is being written, is being redesigned and has a much

[1] At the time the first version of this text was drafted, what it said there was factual; but in the spring of 2020 the ConTeXt wiki was updated and from then on we have to assume that the "official" distribution of ConTeXt has become LMTX. However, for those coming to the world of ConTeXt for the first time, I would still recommend using "ConTEXt Standalone" since it is a more stable distribution. Appendix A explains how to install either distribution.

clearer structure, although it too mixes explanations that only work in Mark II with others for Mark IV or for both versions. This lack of differentiation is also found in the official list of ConTEXt commands[1] which does not specify which commands work only in one of the two versions.

Basically, this introduction has been written by drawing from the four information sources listed here: The ConTEXt "Excursion", the 2013 manual, the contents of the wiki and the official list of commands that includes, for each of them, the allowable configuration options; in addition, of course, my own tests and conclusions. So, in fact this introduction is the result of an investigative effort, and for some time I was tempted to call it "What I know about ConTEXt Mark IV" or "What I have learned about ConTEXt Mark IV". Ultimately, I discarded these titles because, as true as they may be, I felt that they might dissuade someone from getting into ConTEXt; and what is certain is that although the documentation has (in my view) some shortcomings, here we have a truly useful and versatile tool for which the effort it takes to learn it is undoubtedly worthwhile. By using ConTEXt we can manipulate and configure text documents to achieve things that someone who does not know the system simply cannot even imagine.

> Because of who I am, I cannot help the fact that my complaints about the lack of information will appear from time to time throughout this document. I would not like this to be misunderstood: I am immensely grateful to the creators of ConTEXt for having designed such a powerful tool and for having made it available to the public. It is simply that I cannot avoid thinking that this tool would be much more popular if its documentation were improved: one has to invest a lot of time into learning it, not so much because of its intrinsic difficulty (which it has, but no greater than other similar tools – to the contrary in fact), but due to the lack of clear, complete and well-organised information that differentiates between the two versions of ConTEXt, explaining the functions in each of them and, above all, clarifying what each command, argument and option does.
>
> It is true that this kind of information demands great time investment. But given that many commands share options with similar names, perhaps a kind of *glossary* of options could be provided that would also help to detect some inconsistencies resulting from when two options with the same name do different things, or when, to do the same thing, one uses the names of different options in different commands.
>
> As for the reader who is approaching ConTEXt for the first time, let my complaints not dissuade you, because although it may be true that deficient information increases the time needed to learn it, at least for the material dealt with in this introduction I have already invested this time so that the reader does not have to do so. And just with what can be learned from this introduction, readers will have at their disposal a tool that will allow them to produce documents with an ease that they could never have suspected.

Since what is explained in this document comes to a large extent from my own conclusions, it is likely that even though I have personally tested most of what I explain, some statements or opinions may be neither correct nor very orthodox. I

[1] For the list see section 3.6.

will, of course, appreciate any correction, nuance or clarification readers can offer me, and these can be sent to joaquin@ataz.org. However, to reduce the occasions where I am likely to be wrong I have tried not to enter into matters about which I have found no information and that I have not been able (or have not wanted) to personally try out. At times this is the case because the results of my tests were not conclusive, and at other times because I have not always tested everything: the number of commands and options ConTEXt has is impressive, and if I had to try everything out I would never have finished this introduction. There are occasions, however, when I cannot avoid *assuming* something, i.e. making a statement that I see as probable but that I am not completely sure about. In these cases, a 'conjecture' image has been placed in the left margin of the paragraph where I am making such an assumption. The image  aims to graphically represent the assumption.[1] At other times, I have no choice but to admit that I don't know something and I don't even have a reasonable assumption about it: in this case, the image visible to the immediate left in the margin is meant to represent more than just conjecture or ignorance.[2] But as I have never been very good with graphic representations, I am not sure that the images I have selected really manage to convey so many nuances.

This introduction, on the other hand, has been written from the point of view of a reader who knows nothing about either TEX or ConTEXt, although I hope that it can also be useful to those coming from TEX or LATEX (the most popular of the TEX derivatives) who are approaching ConTEXt for the first time. Just the same, I am aware that in trying to please so many different kinds of reader, I run the risk of satisfying nobody. Therefore, in case of doubt, I have always been clear that the principal addressee of this document is the newcomer to ConTEXt, the newcomer who has just come to this fascinating ecosystem.

Being a newcomer to ConTEXt does not imply also being a newcomer to using computer tools; and although in this introduction I am not assuming any particular level of computer literacy in readers, I do presume a certain "reasonable literacy" that implies, for example, having a general understanding of the difference between a word processor and a text editor, knowing how to create, open and manipulate a text file, knowing how to install a program, knowing how to open a terminal and execute a command... and little else.

> Reading through the previous parts of this introduction as I write these lines, I realise that sometimes I get carried away and get into computer issues that are not necessary for learning ConTEXt and that could scare the newcomer off, while at other times I am busy explaining quite obvious things that could bore the experienced reader. I beg the indulgence of both. Rationally, I know that it is very difficult for a complete beginner in computerised text management to even

[1] I did not draw the image, but downloaded it from the internet (https://es.dreamstime.com/), where it says that it is a royalty-free image.

[2] Also found on the internet (https://www.freepik.es/) where its free use is authorised.

> know that ConTEXt exists, but from another point of view, in my professional environment I am surrounded by people who are constantly struggling with texts when they used word processors, and they do so reasonably well, but never having worked with text files as such they ignore such basic issues as, for example, what encoding text files use or what the difference is between a word processor and a text editor.

The fact that this manual is designed for people who know nothing about ConTEXt or TEX, implies that I have included information that clearly is not about ConTEXt but TEX; but I have understood that it is not necessary to burden readers with information that is not relevant for them, as could be the case if a certain command that *in fact* works, is really a ConTEXt command or belongs to TEX; so only on some occasions, when it seems to me to be useful, do I clarify that certain commands really belong to TEX.

With regard to the organisation of this document, the material is grouped into three blocks:

- **The first part**, comprising the first four chapters, offers a global overview of ConTEXt, explaining what it is and how we work with it, showing a first example of how to transform a document so as to be able, later, to explain some fundamental concepts of ConTEXt along with certain questions relating to ConTEXt source files.

  As a whole, these chapters are intended for readers who up until now have only known how to work with word processors. A reader who already knows about working with markup languages could forgo these early chapters; and if the reader already knows TEX, or LATEX, they could also skip much of the content in Chapters 3 and 4. Just the same, I would recommend at least reading:

  – The information relating to ConTEXt commands (Chapter 3), and in particular how it functions, how it is configured, because this is where the principal difference lies between the conception and syntax of LATEX and ConTEXt. Since this introduction refers only to the latter, these differences are not expressly indicated as such, but someone reading this chapter who knows how LATEX works will immediately understand the difference in syntax of the two languages, as also the way that ConTEXt allows us to configure and customise the way almost all of its commands work.

  – The information relating to multifile ConTEXt projects (Chapter 4), which is not so similar to the way of working with other TEX-based systems.

- **The second part**, that includes Chapters 5 to 9, focuses on what we consider to be the main global aspects of a ConTEXt document:

  – The two aspects that mainly affect the appearance of a document are the size and layout of its pages and the font used. Chapters 5 and 6 are dedicated to these matters.

  - ⋆ The first focuses on pages: size, the elements that make up a page, its layout (meaning, how the page elements are distributed), etc. For systematic reasons, more specific aspects are also dealt with here, such as those relating to pagination and the mechanisms that allow us to influence it.

  - ⋆ Chapter 6 explains commands related to the fonts and their handling. Also included here is a basic explanation of the use and management of colours since, although these are not strictly a *characteristic* of fonts, they are just as much an influence on the external appearance of the document.

- Chapters 7 and 8 focus on the structure of the document and the tools that ConTEXt makes available to the author for writing well-structured documents. Chapter 7 focuses on structure properly so called (structural divisions of the document) and Chapter 8 on how this is reflected in the Table of Contents; although, in line with the explanation of this, we use the opportunity to also explain how to generate various kinds of indexes with ConTEXt, since for ConTEXt these all come under the notion of "lists".

- Finally, Chapter 9 focuses on references, an important global aspect of any document when we need to refer to something in another part of the document (internal references) or to other documents (external references). In the case of the latter, we are only interested for the moment in references (links) that mean going to an external document. These *links* (that can also occur in internal references) make our document *interactive*, and in this chapter we explain some of the features of ConTEXt for creating these kinds of documents.

These chapters do not need to be read in any particular order except for Chapter 8, which may be easier to understand if Chapter 7 has been read first. In any case, I have tried to ensure that when a question arises in a chapter or section that is dealt with elsewhere in this introduction, the text includes a mention of that together with a hyperlink to the point where the question is dealt with. However, I am not in a position to guarantee that this will always be the case.

- Finally **the third part** (Chapters 10 and following) focuses on more detailed aspects. They are independent not only of each other, but even of their sections (except, perhaps, in the last chapter). Given the large number of utilities that ConTEXt incorporates, this part could be very extensive; but since my understanding is that by the time they arrive here readers will already be prepared to dive into ConTEXt documentation of their own accord, I have only included the following chapters:

  - Chapters 10 and 11 deal with what we could call the *core elements* of any text document: the text is made up of characters which make up words that are grouped on lines, which in turn make up paragraphs separated from one another by vertical space... Clearly, all these issues could have been included in a single chapter, but as this would be too long, I have divided this matter into two chapters, one that deals with characters, words and horizontal space and another that deals with lines, paragraphs and vertical space.

  - Chapter 12 is a kind of *mishmash* dealing with elements and constructions commonly found in documents; for the most part academic or scientific or technical documents: footnotes, structured lists, descriptions, numbering, etc.

  - Finally, Chapter 13 focuses on floating objects especially the most typical of these: images inserted into documents, and tables.

- The introduction closes with three **appendices**. One is about installing ConTEXt, a second appendix contains several dozen commands that allow the generation of various symbols – mainly but not only for mathematical use, and a third appendix contains an alphabetical list of ConTEXt commands explained or mentioned in the course of this text.

There are many issues that remain to be explained: dealing with quotes and bibliographic references, writing specialised texts (maths, chemistry...), the connection with XML, the interface with Lua code, modes and processing based on modes, working with MetaPost for designing graphics, etc. This is why, since I am not including a complete explanation of ConTEXt, nor am I pretending to do so, I have called this document "An introduction to ConTEXt Mark IV"; and I have added the fact that the introduction is none too short, because obviously this is the case: a text that has left so many things still in the pipeline but that has already gone beyond 300 pages is not by any means a short introduction. This is because I want the reader to understand the logic of ConTEXt, or at least the logic as I have understood it. It does not claim to be a reference manual, but rather a guide for self-learning that prepares the reader to produce documents of medium complexity (and this includes most of the likely documents) and that above all teaches the reader to *imagine* what can be done with this powerful tool and find out how to do it in the documentation available. Nor is this document a *tutorial*. Tutorials are designed to progressively increase the level of difficulty, so that what is to be learned is taught step by step; in this respect I have preferred to begin with the second part instead of ordering material according to the level of difficulty, in order to be more systematic. But while it is not a tutorial, I have included very many examples.

It is possible that for some readers this document's title reminds them of a text written by Oetiker, Partl, Hyna and Schlegl available on the internet and one of the better documents for introducing oneself to the LaTeX world. I am talking about "*The Not So Short Introduction to LaTeX* 2_ϵ". This is no coincidence, but a tribute and act of appreciation: thanks to the generous work of those who write texts like that, it is possible for many people to begin to work with useful and powerful tools like LaTeX and ConTeXt. These authors helped me to start out with LaTeX; I am hoping to do the same with someone who wants to start out with ConTeXt, even though in the original Spanish version of this text I stuck exclusively to the Spanish-speaking world who have lacked so much documentation in their language. I hope this document fulfils this expectation, and in the meantime, others have generously offered to translate it into other languages, hence this English edition. Thank you.

Joaquín Ataz-López
Summer 2020

I
What is ConTEXt and how do we work with it

Chapter 1
ConTEXt: a general overview

1.1 What is ConTEXt then?

ConTEXt is a *typesetting system*, or in other words: an extensive set of tools aimed at giving the user absolute and complete control over the appearance and presentation of a specific electronic document intended for print on paper or to be shown on screen. This chapter explains what this means. But first, let us highlight some of the characteristics of ConTEXt.

- There are two *flavours* of ConTEXt known as Mark II and Mark IV respectively. ConTEXt Mark II is frozen, i.e. it is considered to be an already fully-developed language that is not intended to have further changes or new things added. A new version would appear only in the case where some error needs to be corrected. ConTEXt Mark IV, on the other hand, continues to be developed so that new versions appear from time to time that introduce some improvement or additional utility. But, although still in development, it is a very mature language in which changes introduced by the new versions are quite subtle and exclusively affect the low level operation of the system. For the average user these changes are totally transparent; it is as if they did not exist. Although both *flavours* have much in common, there are also some incompatible features between them. Hence this introduction focuses only on ConTEXt Mark IV.

- ConTEXt is software *libre* (or free software, but not just in the sense of *gratis*). The program properly speaking (that is, the complex of computer tools that make up ConTEXt), is distributed under the *GNU General Public Licence*. The documentation is offered under the "*Creative Commons*" licence that allows it to be freely copied and distributed.

- ConTEXt is neither a word processor program nor a text editing program, but a collection of tools aimed at *transforming* a text we have previously written in our favourite text editor. Therefore, when we work with ConTEXt:
  - We begin by writing one or more text files with any kind of text editor.
  - In these files, along with the text that makes up the contents of the document, there is a range of instructions that tell ConTEXt about the appearance that the final document generated from the original text files must have. The complete set of ConTEXt instructions, in fact, is a *language*; and since this language allows one to *program* the typographical transformation of a text, we can say that ConTEXt is a *typographical programming language*.
  - Once we have written the source files, these will be processed by a program (also called "`context`"[1]), which will generate a PDF file from them ready to be sent to a print shop or to be shown on screen.
- In ConTEXt, therefore, we must differentiate between the document we are writing, and the document that ConTEXt generates. To avoid any doubts, in this introduction I will call the text document that contains formatting instructions the *source file*, and the PDF document generated by ConTEXt from the source file I will call the *final document*.

The above basic points will be further developed below.

1.2 Typesetting texts

Writing a document (book, article, chapter, leaflet, print out, paper ...) and putting it together typographically are two very different activities. Writing the document is much the same as drafting it; this is done by the author who decides on its content and structure. The document created directly by the author, just as he or she wrote it, is called the *manuscript*. By its very nature, only the author or those permitted to read it have access to the manuscript. Its dissemination beyond this intimate group requires the manuscript to be *published*. Today, publishing something – in the etymological sense of making it "accessible to the public" – is as simple as putting it on the internet, available to anyone who finds it and wants

[1] ConTEXt is both a language and a program at the same time (besides being other things). This fact, in a text like the current one, poses the problem that at times we have to distinguish between these two aspects. This is why I have adopted the typographical convention of writing "ConTeXt" with its logo (ConTEXt) when I want to refer exclusively to the language, or to both the language and the program. However, when I want to refer exclusively to the program, I will write "`context`" all in lower case and in the monospaced type that is typical of computer terminals and typewriters. I will also use this type for examples and mentions of commands belonging to this language.

to read it. But until relatively recently, publication was a cost-intensive process dependent on certain professionals specialised in it, only accessed by those manuscripts which, because of their content, or because of their author, were considered to be particularly interesting. And even today we tend to reserve the word *publication* for this kind of *professional publication* where the manuscript undergoes a series of transformations in its appearance aimed at improving the *legibility* of the document. This series of transformations is what we call *typesetting*.

The aim of typesetting is – generally speaking, and leaving aside advertisement-type texts that seek to attract the reader's attention – to produce documents with the greatest *legibility*, meaning the quality of the printed text that invites or facilitates its reading and ensures that the reader feels comfortable with it. Many things contribute to this; some, of course, have to do with the document's *contents*: (quality, clarity, organisation...), but others depend on things like the type and dimensions of the font used, the use of white space in the document, visual separation between paragraphs, etc. In addition, there are other kinds of resources, not so much of the graphic or visual kind, such as the presence or otherwise in the document of specific aids to the reader like page headers and footers, indexes, glossaries, use of bold type, margin headings, etc. The knowledge and correct handling of all the resources available to a typesetter could be called the “art of typesetting” or the “art of printing”.

Historically, and until the advent of the computer, the tasks and roles of writer and typesetter were kept quite distinct. The author wrote by hand or on a 19th century machine called a typewriter, the typographical resources of which were even more limited than those who wrote by hand; and then the writer gave the originals to the publisher or printer who transformed them to obtain the printed document.

Today, computer science has made it easier for the author to decide on the composition down to the last detail. However, this does not do away with the fact that the qualities that a good author needs are not the same as those needed by a good typesetter. Depending on the kind of document being dealt with, the author needs an understanding of the subject matter being written about, clarity of exposition, well-structured thinking that allows for the creation of a well-organised text, creativity, a sense of rhythm, etc. But the typesetter has to combine a good knowledge of the conceptual and graphical resources at his or her disposal, and sufficient good taste to be able to use them harmoniously.

With a good word processing program[1] it is possible to achieve a reasonably good typographically prepared document. But word processors, generally speaking, are

[1] According to a rather old convention, we make a distinction between *text editors* and *word processors*. The early kinds of text editing programs dealt with unformatted text files, while the other kind worked with binary files of formatted text.

not designed for typesetting and the results, although they may be correct, are not comparable to the results obtainable with other tools designed specifically to control the composition of the document. In fact, word processors are how typewriters evolved, and their use, to the extent that these tools mask the difference between the authorship of the text and its typesetting, tends to produce unstructured and typographically inadequate texts. On the contrary, tools like ConTEXt have evolved from the printing press; they offer many more composition possibilities and above all, it is not possible to learn how to use them without also acquiring, along the way, many notions relating to typesetting. This is the difference from word processors, which someone can use for many years without learning a single thing about typography.

1.3 Markup languages

In the days before computers, as I said before, the author prepared the manuscript by hand or typewriter and handed it to the publisher or printer who was responsible for the transformation of the manuscript into the final printed text. Although the author had relatively little involvement in the transformation, he or she did maintain some intervention by pointing out, for example, that certain lines of the manuscript were the titles of its various parts (chapters, sections ...), or by indicating that certain things should be highlighted typographically in some way. These indications were made by the author in the manuscript itself, sometimes expressly, and at other times through certain conventions that continued to develop over time. For example, the chapters always began on a new page by inserting several blank lines before the title, underlining it, writing it in capital letters, or framing the text to be highlighted between two underscores, increasing the indentation of a paragraph, etc.

To put it briefly, the author *marked up* the text in order to provide indications relating to how it should be typeset. Then later, the editor would handwrite other indications on the text for the printer, such as, for example, the font to be used, and its size.

Today, in a computerised world, we can continue to do this to generate electronic documents through what is called a *markup language*. These kinds of languages use a series of *marks* or indications that the program processing the file containing them knows how to interpret. Probably the best known markup language today is HTML, since most web pages today are based on it. An HTML page contains the text of a web page, along with a series of marks that tell the browser program that loads the page how it should display it. The HTML markup that web browsers understand, together with instructions about how and where to use them, is called the “HTML language”, which is a *markup language*. But as well as HTML, there are many other markup languages; in fact they are booming, and so XML, which

is the markup language *par excellence*, is found everywhere today and is in use for pretty much everything: for database design, for the creation of specific languages, transmission of structured data, application configuration files, etc. There are also markup languages intended for graphic design (SVG, TikZ or MetaPost), maths formulas (MathML), music (Lilypond and MusicXML), finance, geomatics, etc. And of course there are also markup languages aimed at the typographical transformation of text, and among these, TEX and its derivatives stand out.

With regard to the *typographical* markup that indicates how a text should look, there are two kinds that we can refer to: *purely typographical markup* and *conceptual markup* or, if you prefer, *logical markup*. Purely typographical markup is limited to indicating precisely what typographical resource should be used to display a certain text; such as when, for example, we indicate that certain text should be in bold or italics. Conceptual markup, on the other hand, indicates what function complies with certain text in the document as a whole, such as when we indicate that something is a title, or a subtitle, or a quote. In general, documents that prefer to use this second kind of markup are more consistent and easier to compose, since once again they point out the difference between authorship and composition: the author indicates that such and such a line is a title, or that such and such a fragment is a warning, or a quote; and the typesetter decides how to typographically highlight all titles, warnings or quotations; thus, on the one hand, consistency is guaranteed, as all the fragments that fulfil the same function will look the same, and, on the other hand it saves time, because the format of each type of fragment only needs to be indicated once.

1.4 TEX and its derivatives

TEX was developed towards the end of the 70s by Donald E. Knuth, a professor (now emeritus professor) of theoretical computer programming at Stanford University, who implemented the program to produce his own publications and as an example of a systematically developed and annotated program. Along with TEX, Knuth developed an additional programming language called MetaFont, created for designing typographical fonts, and he used it to design a font he baptised as *Computer Modern*, which, along with the usual characters of any font, also included a complete set of "glyphs"[1] designed for writing mathematics. To all this he added some additional utilities and thus the typesetting system called TEX was born, which, due to its power, quality of results, flexibility of use and broad possibilities, is considered one of the best computerised systems for text composition.

[1] In typography, a glyph is the graphical representation of a character, a number of characters or part of a character, and is today's equivalent of the letter type (the bit engraved with the letter or movable type).

It was designed for texts in which there was a lot of mathematics, but it soon became clear that the system's possibilities made it suitable for all kinds of texts.

Internally, TEX functions in the same way as the former compositors would do in a print shop. For TEX, everything is a *box*: The letters are contained in boxes, the blank spaces are also boxes, several letters (the boxes containing several letters) form a new box that encloses the word, and several words, along with the blank space between them, form a box containing a line, several lines become a box containing the paragraph ... and so on. All this, moreover, with extraordinary precision in the handling of measurements. Consider that the smallest unit that TEX deals with is 65.536 times smaller than the typographical point with which characters and lines are measured, which is usually the smallest unit handled by most word processing programs. This means that the smallest unit handled by TEX is approximately 0.000005356 millimetres.

The name TEX comes from the root of the Greek word τεχνη, written in upper case letters (TÉXNH). Therefore, the final letter of the word TEX is not a Latin 'X', but the Greek 'χ', pronounced – apparently – like the Scottish 'ch' in *loch*. So TEX should be pronounced as *Tech*. This Greek word, on the other hand, meant both "art" and "technology", and this is the reason why KNUTH chose it to name his system. The purpose of this name – he wrote – "is to remind you that TEX is primarily concerned with high quality technical manuscripts. Its emphasis is on art and technology, as in the underlying Greek word".

Using the convention established by KNUTH, TEX is to be written:

- In typographically formatted texts like this one, using the logo that I have been using until now: the three letters in upper case, with the central 'E' slightly displaced below to facilitate a closer alignment between the 'T' and the 'X'; or in other words: "TEX".

  > To facilitate the writing of this logo, Knuth included an instruction in TEX for writing it in the final document: `\TeX`.

- In unformatted texts (such as an email, or a text file), with the 'T' and the 'X' in upper case, and the central 'e' in lower case; so: "TeX".

This convention continues to be used in all derivatives of TEX that include its proper name, as is the case with ConTEXt. When writing it in text mode we need to write "ConTeXt".

1.4.1 TEX engines

The TEX program is free *libre* software: its source code is available to the public and anyone can use it or modify it as they wish, with the only condition that, if modifications are made, the result cannot be called "TEX". This is why, over time,

certain adaptations of the program have emerged, introducing different improvements to it, and which are generally referred to as *TEX engines*. Apart from the original TEX program, the main engines are, in chronological order of appearance, pdfTEX, ε-TEX, XETEX and LuaTEX. Each of them is supposed to incorporate the improvements of the previous one. These improvements, on the other hand, up until the appearance of LuaTEX, did not affect the language itself, but only the input files, the output files, handling of sources and low level operation of macros.

> The question of which TEX engine to use is a much debated one within the TEX universe. I will not develop this question here since ConTEXt Mark IV only works with LuaTeX. In reality, in the ConTEXt world, discussion on TEX *engines* becomes a discussion on whether to use Mark II (that works with PdfTeX and XeTeX) or Mark IV (that only works with LuaTeX).

1.4.2 Formats derived from TEX

The core or heart of TEX only understands a set of approximately 300 very basic instructions, called *primitives*, which are suitable for typesetting operations and programming functions. The great majority of these instructions are of a very *low level*, which, in computer terminology, means that they are more easily understandable by the computer than by human beings, since they concern very elementary operations of the "shift this character 0.000725 millimetres upward" kind. Hence KNUTH saw that TEX would be extensible, meaning that there should be a mechanism that allows instructions to be defined at a higher level, more easily understandable by human beings. These instructions, that are broken down into other simpler instructions at the time of execution, are called *macros*. For example, the TEX instruction that prints the (`\TeX`) logo, is broken down as follows at the time of execution:

```
T
\kern -.1667em
\lower .5ex
\hbox {E}
\kern -.125em
X
```

But for the human being, it is much easier to understand and remember that the simple command "`\TeX`" carries out the typographical operations needed to print the logo.

> The difference between what is a *macro* and what is a *primitive*, really only has importance from the perspective of the TEX developer. From the user's perspective they are *instructions* or, if you prefer, *commands*. Knuth called them *control sequences*.

This possibility of extending the language through *macros* is one of the characteristics that turned TEX into such a powerful tool. In fact, KNUTH himself created approximately 600 macros that, along with the 300 primitives, make up the format

called "Plain TEX". It is quite common to confuse TEX properly so called, with Plain TEX and, in fact, almost everything usually written or said about TEX, is really a reference to Plain TEX. Books that claim to be about TEX (including the foundational "*The TEXBook*"), really refer to Plain TEX; and those who believe they are directly working with TEX are in reality working with Plain TEX.

Plain TEX is what, in TEX terminology, is called a *format*, consisting of a broad set of macros, together with certain rules of syntax concerning how and in what way to use them. As well as Plain TEX, with the passing of time other *formats* have been developed, among which it is worth mentioning LATEX, a broad set of macros for TEX created in 1985 by Leslie Lamport and which is probably the TEX derivative that is most in use in the academic, technological and mathematical world. ConTEXt is (or has begun to be), on a par with LATEX as a format derived from TEX.

Normally these *formats* are accompanied by a programme that loads the macros that make them up into memory before calling on "`tex`" (or the actual engine being used for processing) to process the source file. But even though all these formats are actually running TEX, as each of them has different instructions and different syntax rules from the user's point of view, we can think of them as *different languages*. They all draw their inspiration from TEX, but are different from TEX and also different from each other.

1.5 ConTEXt

In reality ConTEXt, which started out as a *format* of TEX, is much more than that today. ConTEXt includes:

1. A very broad set of TEX macros. If Plain TEX has around 900 instructions, ConTEXt has around 3500; and if we add up the names of the different options that these commands support, we are talking about a vocabulary of around 4000 words. The vocabulary is this large because of the ConTEXt strategy to facilitate its learning, and this strategy means the inclusion of any number of synonyms for commands and options.

   > The intention is that if a certain effect is to be achieved, then for each of the ways an English speaker would call that effect there is a command or option that achieves it – which is supposed to make the use of the language easier. For example, to simultaneously get a bold and italic letter, ConTEXt has three instructions all of which achieve the same result: `\bi`, `\italicbold` and `\bolditalic`.

2. A likewise broad set of macros for MetaPost, a graphical programming language derived from MetaFont, which in turn is a language for typeface design that Knuth developed jointly with TEX.

3. Various *scripts* developed in PERL (the oldest), RUBY (some also old, others not so old) and LUA (the most recent).

4. An interface that integrates TEX, MetaPost, LUA and XML, allowing one to write and process documents in any of these languages, or to mix elements from some of them.

   Perhaps you did not understand much of the previous explanation? Don't worry about it. I used a lot of computer jargon in it and mentioned many programs and languages. It is not necessary to know all the different components to use ConTEXt. The important thing, at this stage of learning, is to stay with the idea that ConTEXt integrates many tools from different sources that together make up a *typesetting system*.

It is because of this latter feature of integration of tools with different origins, that we say that ConTEXt is a "hybrid technology" intended for typesetting documents. My understanding is that this turns ConTEXt into an extraordinarily advanced and powerful system.

Even though ConTEXt is much more than a collection of macros for TEX, it continues to be based on TEX, and this is why this document, that I claim to be no more than an *introduction*, focuses on this.

ConTEXt, on the other hand, is rather more modern than TEX. When TEX was created, the emergence of computers was just at the beginning, and we were far from seeing what the internet and the multimedia world would be (would become). In this respect, ConTEXt naturally integrates some of the things that have always been something of a foreign body in TEX such as including external graphics, handling colour, hyperlinks in electronic documents, assuming a paper size suitable for a document intended for display on a screen, etc.

1.5.1 A short history of ConTEXt

ConTEXt was born approximately in 1991. It was created by HANS HAGEN and TON OTTEN in a Dutch document design and processing company called "*Pragma Advanced Document Engineering*", usually abbreviated as Pragma ADE. It began by being a collection of TEX macros that had Dutch names and was unofficially known as *Pragmatex*, aimed at the company's non-technical employees who had to manage the many details of editing typeset documents and who were not used to using markup languages or interfaces other than Dutch. Hence the first version of ConTEXt was written in Dutch. The idea was to create a sufficient number of macros with a uniform and consistent interface. Approximately in 1994 the *package* was stable enough for a user manual to be written in Dutch, and in 1996, through the initiative of HANS HAGEN, reference to it began taking on the name "ConTEXt". This name claims to mean "Text with TEX" (using the Latin preposition 'con' meaning 'with'), but at the same time a wordplay on the English (and Dutch)

word “Context”. Behind the name, therefore, lies a triple play on words involving “TEX”, “text” and “context”.

> Therefore, since the name is based on wordplay, ConTEXt should be pronounced ‘context’ and not ‘contecht’ since this would mean losing the play on words.

The interface began to be translated into English approximately in 2005, giving rise to the version known as ConTEXt Mark II, where the ‘II’ is explained because in the mind of the developers, the previous version in Dutch was Version ‘I’, even though it was never officially called that. After the interface was translated into English, the use of the system began to spread beyond the Netherlands, and the interface was translated into other European languages such as French, German, Italian and Romanian. The “official” documentation for ConTEXt, nevertheless, is normally based on the English version, and this is the version this document works with.

In its initial version, ConTEXt Mark II worked with the PdfTeX *TEX engine*. But later, at the appearance of the XeTEX *engine*, ConTEXt Mark II was modified to allow the use of this new engine that contributed a number of advantages by comparison with PdfTeX. But when LuaTEX came along some years later, the decision was made to internally reconfigure how ConTEXt functioned in order to integrate all the new possibilities that this new engine offered. And so, ConTEXt Mark IV was born, and it was presented in 2007, immediately after the presentation of LuaTEX. Very probably, one of the influencing factors in the decision to reconfigure ConTEXt to adapt it to LuaTEX was that two of the three main developers of ConTEXt, Hans Hagen and Taco Hoekwater, were also part of the main team developing LuaTEX. This is why ConTEXt Mark IV and LuaTEX were born at the same time and developed in unison. There is a synergy between ConTEXt and LuaTEX that does not exist in any other derivative of TEX; and I doubt that any of the others can avail themselves of the advantages of LuaTEX as ConTEXt can.

There are many differences between Mark II and Mark IV, although most of them are *internal*, that is, they have to do with how the macro actually works at a lower level, such that from the user's perspective the differences are not noticeable: the name and parameters of the macro remain the same. There are, however, some differences that affect the interface and force one to do things differently depending on which version one is working with. These differences are relatively few, but they do affect very important aspects such as for example, the coding of the input file, or the handling of fonts installed in the system.

> It would, however, be very welcome if somewhere there were a document that explained (or listed) the appreciable differences between Mark II and Mark IV. In the ConTEXt wiki, for example, for each ConTEXt command there are *two kinds of syntax* (very often identical). I presume one belongs to Mark II and the other to Mark IV; and based on this assumption, I also

presume that the *first version* is from Mark II. But the truth is that the wiki tells us nothing about this.

The fact that the differences, at a language level, are relatively few, means that on many occasions rather than speaking of two versions we are talking about two "flavours" of ConTEXt. But whether you call them one or the other, the fact is that a document prepared for Mark II cannot normally be compiled with Mark IV and vice versa; and if the document mixes both versions, it will most likely not compile well with either of them; which implies that the author of the source file has to start by deciding whether to write for Mark II or for Mark IV.

If we work with the different versions of ConTEXt, a good trick for differentiating at first sight between files intended for Mark II and those intended for Mark IV is to use a different extension for the file names. Thus, for example, for any files I have written for Mark II, I put "`.mkii`" as the extension, and "`.mkiv`" instead for those written for Mark IV. It is true that ConTEXt expects all source files to have the extension "`.tex`", but we can change the file extension as long as we expressly indicate the file extension when applying ConTEXt to the file.

The ConTEXt distribution installed on the wiki, "ConTEXt Standalone", includes both versions, and to avoid confusion – I assume – uses a different command for each of them to compile a file. Mark II compiles with the command "`texexec`" and Mark IV with the command "`context`".

In fact both commands, "`context`" and "`texexec`", are *scripts* with different options that run "`mtxrun`", which in turn is a Lua *script*.

Today, Mark II is frozen and Mark IV continues to be developed, which means that new versions of the former are only published when errors or faults are discovered that need to be corrected, while new versions of Mark IV continue to be published regularly; sometimes two or three times a month, even though in most of these cases the "new versions" do not introduce perceptible changes in the language but are limited to somehow improving implementation of a command at low level, or updating some of the many manuals included with the distribution. Even so, if we have installed the development version – which is what I would recommend and which is the one installed by default with "ConTEXt Standalone" – it makes sense to update our version from time to time (See Appendix A for the way of updating the installed version of "ConTEXt Standalone").

LMTX and other alternative implementations of Mark IV

The developers of ConTEXt are naturally restless, and therefore have not ceased development of ConTEXt with Mark IV; new versions are still being tested and experimented with, although in general these differ from Mark IV in very few ways, and do not have the incompatibility in compiling that exists between Mark IV and Mark II.

Thus, certain minor variants of Mark IV called, respectively, Mark VI, Mark IX and Mark XI have been developed. Of these, I have only been able to find a small reference to Mark VI in the ConTEXt wiki where it says that the only difference with Mark IV lies in the possibility of

defining commands by assigning the parameters not a number, as is traditional in TEX, but a name, as is usually done in almost all programming languages.

More important than these small variations, I believe, is the appearance in the ConTEXt universe (ConTEXtverse?) of a new version called LMTX, a name which is an acronym of LuaMetaTeX: a new TEX *engine* that is a simplified version of LuaTEX, developed with a view to saving computer resources; which means that LMTX requires less memory and less processing power than ConTEXt Mark IV.

LMTX was presented in spring 2019 and one assumes that it will not imply any external change to the Mark IV language. For the author of the document there would be no difference at the time of working with it; but when compiling it, one would need to choose between doing so with LuaTEX, or doing so with LuaMetaTeX. In Appendix A, relating to the installation of ConTEXt, a procedure is shown for assigning a different command name to each of the installations (section 3).

1.5.2 ConTEXt versus LATEX

Given that the most popular format derived from TEX is LATEX, a comparison between this and ConTEXt is inevitable. Clearly we are talking about different languages although in some way related to each other since they both derive from TEX; the relationship is similar to that which exists, for example, between Spanish and French: languages that have a common origin (Latin) which means that their syntax is *similar* and many of the words in each of these languages is mirrored by a word in the other. But apart from this *family resemblance*, LATEX and ConTEXt differ in their philosophy and implementation, since the initial aims of both, are, to some degree, the opposite. LATEX claims to facilitate the use of TEX, isolating the author from the concrete typographical details to help focus on content, leaving the typesetting details in the hands of LATEX. This means that simplifying the use of TEX takes place at the expense of limiting the immense flexibility of TEX, by predefining basic formats and limiting the number of typographical issues that the author has to decide on. In contrast to this philosophy, ConTEXt was born within a company dedicated to typesetting documents. Therefore, far from wanting to isolate the author from typesetting details, the aim is to give the author absolute and complete control over them. To achieve this, ConTEXt provides a uniform and consistent interface which is much closer to the original spirit of TEX than LATEX.

This difference in philosophy and founding objectives then translates, in turn, into a difference in implementation. LATEX, that tends to simplify things as much as possible, does not need to use all of TEX's resources. In some way, its core is rather simple. So when there is a need to broaden its possibilities, it is necessary to expressly write a *package* to do so. This *packaging* associated with LATEX is both a virtue and a defect: a virtue, because the tremendous popularity of LATEX, together with the generosity of its users, means that almost any need we are likely to have has been met by someone before, and that there is a package to achieve it; but it is also a defect because these packages are often incompatible with each other, and

their syntax is not always uniform. This means that working with LaTeX requires one to constantly be searching through thousands of already existing packages to fulfil one's needs and ensure that they all work together.

By contrast with the simplicity of the LaTeX core, which is complemented by its extensibility through packages, ConTEXt is designed to have within it all – or almost all – the typographical possibilities of TEX, so its conception is much more monolithic, but at the same time it is also more modular. The ConTEXt core allows us to do almost everything, and we are guaranteed that there will be no incompatibilities between its different utilities, no need to investigate extensions for what we need, and the syntax of the language does not change just because we need a particular utility.

It is true that ConTEXt has what are called extension *modules* that some might consider as carrying out a function similar to the LaTeX packages, but in real terms they both work differently: ConTEXt modules are designed exclusively to include additional utilities that, because they are still in an experimental stage, have not yet been incorporated into the core, or to allow access to extensions authored by someone outside the ConTEXt development team.

I do not believe that either one of these two *philosophies* is preferable to the other. The question depends rather on the user's profile and what he or she wants. If the user does not want to deal with typographical issues but simply produce very high quality standardised documents, it would probably be preferable to opt for a system like LaTeX; on the other hand, the user who likes to experiment, or who needs to control every last detail of the document, or someone who has to devise a special layout for a document, would probably be better off using a system like ConTEXt, where the author has all the control in their hands; with the risk, of course, of not knowing how to use this control correctly.

1.5.3 A good understanding of the dynamics of working with ConTEXt

When we work with ConTEXt, we always begin by writing a text file (which we call a *source file*), in which, along with the actual content of our final document, we will include the instructions (in ConTEXt-speak) that indicate exactly how we want the document to be formatted: the general appearance we want its pages and paragraphs to have, the margins we want to apply to certain paragraphs, the font we want to display, the snippets we want shown in a different font, etc. Once we have written the source file, we apply the "`context`" program from a terminal, which will process it, and will generate a different file from it in which the contents of our document will be formatted in accordance with the instructions included in the source file for this purpose. This new file could be sent to a (commercial) printer, displayed on screen, placed on the internet or distributed among contacts,

friends, clients, teachers, pupils ... or in other words, to anyone for whom we wrote the document.

This means that when working with ConTEXt the author is working with a file whose appearance has nothing to do with the final document: the file the author is directly working on is a text file that is not formatted typographically. So ConTEXt works in a different way than do programs known as *word processors* that show the final appearance of the edited document at the same time we are writing it. For those accustomed to word processors, the way of working with ConTEXt will initially feel strange, and it may even take some time to get used to it. However, once one gets used to it, one understands that in reality this other way of working, differentiating between the work file and the final result, is actually an advantage for many reasons, among which I will highlight here, without following any particular order, the following:

1. Because text files are 'lighter' to handle than word processor binary files, and editing them requires less computer memory, they are less likely to be corrupted, and they do not become unintelligible when we change the version of the program we are creating them with. They are also compatible with any operating system, and can be edited with many text editors, so that in order to work with them it is not necessary for the computer system to have the program the file was created with installed on it: any other editing program will do; and in every computer system there is always some text editing program.

2. Because differentiating between the working document and the final document helps to distinguish what the actual content of the document is from what its appearance will be, allowing the author to concentrate on the content in the creation phase, and to focus on the appearance in the typesetting phase.

3. Because it allows one to quickly and accurately change the appearance of the document, since this is determined by ConTEXt commands that can be easily identified.

4. Because this facility for changing the appearance, on the other hand, allows us to easily generate two (or more) different versions from a single content: Perhaps one version optimised for printing on paper, and another designed to be displayed on screen, adjusted to the size of the latter and perhaps including hyperlinks that make no sense in a paper document.

5. Because typographical errors (typos) that are common in word processors, such as extending the italics beyond the last character of a word, are also easily avoided.

6. Because while the work file is not distributed and is 'for our eyes only', it is possible to incorporate annotations and observations, comments and warnings

for ourselves for subsequent revisions or versions, with the peace of mind in knowing that these will not appear in the formatted file to be distributed.

7. Because the quality that can be obtained by processing the whole document simultaneously is much higher than that which can be achieved with a program that has to make typographical decisions as the document is being written.

8. Etcetera.

All of the above means that on the one hand when working with ConTEXt, once we have got the hang of it, we are more efficient and productive, and that on the other hand, the typographical quality we can obtain is much superior to what can be obtained with so-called *word processors*. And although it is true that the latter are easier to use, in point of fact they are not that *much* easier to use. Because while it is true that ConTEXt, as we have said before, contains 3500 instructions, a normal user of ConTEXt will not need to know them all. To do what is usually done with word processors, we only need to know the instructions that allow us to indicate the structure of the document, a few instructions concerning common typographical resources, such as bold or italics, and perhaps how to generate a list, or a footnote. In total, no more than 15 or 20 instructions will allow us to do almost all the things that are done with a word processor. The rest of the instructions allow us to do different things that we normally cannot do with a word processor, or are very difficult to achieve. We can say that while learning to use ConTEXt is more difficult than learning to use a word processor, this is because we can do a lot more with ConTEXt.

1.5.4 Getting help with ConTEXt

While we are new to it, the best place for getting help with ConTEXt is, undoubtedly, on the wiki, which abounds in examples and has a good search engine, especially if one understands English well. We can also find help on the internet, of course, but here the play on words in the name ConTEXt will play tricks on us because searching on the word "context" will return millions of results most of which will have nothing to do with what we are looking for. To find information on ConTEXt you need to add something to the word "context"; for example, "tex", or "Mark IV" or "Hans Hagen" (one of the creators of ConTEXt) or "Pragma ADE", or something similar. It could also be useful to seek information using the wiki name: "contextgarden".

When we have learned something more about ConTEXt, we can consult some of the many documents included in "ConTEXt Standalone", or even seek help in TeX – LaTeX Stack Exchange, or on the mailing list for ConTEXt (NTG-context). The latter involves the people who know the most about ConTEXt, but the rules of good cyber-etiquette demand that before asking a question, we should have tried hard beforehand to find the answer ourselves.

Chapter 2
Our first source file

Table of Contents:

This chapter is dedicated to our first experiment, and will explain the basic structure of a ConTEXt document, as well as the best strategies for dealing with potential errors.

2.1 Preparing the experiment: essential tools

To write and compile a first source file, we need the following tools to be installed on our system.

1. **A text editor** for writing our test file. There are many text editors around and it is difficult to think of an operating system that does not already have one installed. We can use any of them: there are simple ones, more complex ones, more powerful ones, some you pay for, some free (as in *gratis*), some free (as in *libre*), some which specialise in TEX systems, others of a general nature, etc. If we are used to handling a particular editor, we would do better to continue working with it; if we are not used to working with one up to this point, my advice, initially, is to find a simple editor so as not to add the task of learning how to use a text editor to the difficulty of learning ConTEXt. Although it is true that often the most difficult programs to learn are the ones that are the most powerful.

   I have written this text with GNU Emacs, which is one of the most powerful and versatile general purpose editors in existence; it is true that it has its peculiarities and also its detractors, but in general there are more "*Emacs-lovers*" than "*Emacs-haters*". There is a GNU Emacs extension called AucTeX for working with TEX files or one of its derivatives, which provides the editor with some very interesting additional utilities, although AucTeX is in general better prepared to work with LATEX than with ConTEXt files. GNU Emacs in combination with AucTeX could be a good option if we don't know which

editor to choose; both are software *libre* programs, and so there are versions of them for all operating systems. In fact, saying that GNU Emacs is *software libre* is an understatement, since this program embodies better than any other the spirit of what *free software* is and means. In the end, its main developer was RICHARD STALLMAN founder and ideologue of the GNU project and the *Free Software Foundation*.

As well as GNU Emacs + AucTeX, other good options, if you do not know which to choose, are *Scite* and *TexWorks*. The former, even though a general purpose editor not specifically designed for working with ConTEXt files, is easily customised and, as it is the editor that ConTEXt developers generally use, "ConTEXt Standalone" contains the configuration files for this editor, written by HANS HAGEN himself. *TexWorks*, on the other hand, is a fast text editor and specialises in handling TEX files and those of its derivative languages. It is quite simple to configure it for working with ConTEXt and "ConTEXt Standalone" also envisages its configuration.

Whatever the editor, the one thing we must not use as a text editor is a *word processor* like, for example, OpenOffice Writer or Microsoft Word. These programs, also too slow and heavy in my opinion can, if it is expressly indicated, save a file as 'text only (txt)', but they were not designed for this and we will most likely end up saving our file in some binary format that is incompatible with ConTEXt.

2. **A ConTEXt** distribution for processing our test file. If there is already a TEX (or LATEX) installation on our system, it is possible that there is already a version of ConTEXt installed. To test this, it is enough to open a terminal and type

```
$> context --version
```

> **NOTE** for those who are new to handling terminals, the first two characters I have written ("$>") do not need to be written in the terminal. I have simply represented what is called the terminal *prompt*; the little blinking sign that indicates that the terminal is awaiting instructions.

If there is already a version of ConTEXt installed, something like the following will appear:

```
mtx-context     | ConTeXt Process Management 1.03
mtx-context     |
mtx-context     | main context file: /home/jq/context/LMTX/tex/texmf-context/
                | tex/context/base/mkiv/context.mkiv
mtx-context     | current version: 2020.04.30 11:15
mtx-context     | main context file: /home/jq/context/LMTX/tex/texmf-context/
                | tex/context/base/mkiv/context.mkxl
mtx-context     | current version: 2020.04.30 11:15
```

The last line informs us of the date when the installed version was released. If this is too old, we should either update it or install a new version. I recommend the installation of the distribution called "ConTEXt Standalone" whose installation instructions can be found on the ConTEXt wiki. You can find a summary of all this in Appendix A .

3. **A reader for PDF files**, so we can see the result of our experiment on screen. In Windows and Mac OS there is always Adobe Acrobat Reader. It is not installed by default (or wasn't when I ceased using Microsoft Windows more than 15 years ago), but it does so the first time you try to open a PDF file so it is most likely that it is already installed. Linux/Unix systems do not have a current version of Acrobat Reader, but nor do you need it since there are literally dozens of free and very good PDF readers available. Besides, there is almost always one of them installed by default on these systems. My favourite, for speed and ease of use, is MuPDF; although it has some drawbacks if you are using languages other than English with accented characters, and it does not allow you to select text or send a document to the printer; it is simply a reader; but it is very fast and comfortable to use. When I need some of the facilities that don't work in MuPDF, I usually use either Okular, or qPdfView. But again, it is a matter of taste: one can choose whatever one prefers.

We can choose our editor, our PDF reader, our ConTEXt distribution ... Welcome to the world of *free libre software*!

2.2 The experiment itself

Writing the source file

If the tools mentioned above are already available, we need to open our text editor and create a file with it that we will call "rain.tex". We will write what follows as the contents of this file:

```
% First line of the document

\mainlanguage[en] % Language = English

\setuppapersize[S5] % paper size

\setupbodyfont
  [modern,12pt] % Font = Latin Modern, 12 point

\setuphead      % Format of chapters
  [chapter]
  [style=\bfc]

\starttext  % Begin document contents

\startchapter
  [title=The rain in Spain...]

How kind of you to let me come.
Now once again, where does it rain?
On the plain, on the plain.
And where's that blasted plain?
In Spain, in Spain.
The rain in Spain stays mainly in the plain.
The rain in Spain stays mainly in the plain.

\stopchapter

\stoptext % End of document
```

While writing it, it does not matter if anything changes, especially if adding or removing white space or line breaks. What is important is that the words following the "\" are written exactly as they are, as well as the contents inside the curly brackets. There can be variations in the rest.

The file's character encoding

Once we have written what is above, we save the file on disk, making sure the character encoding is UTF-8. This character encoding is today's standard. In any case, if we are not sure, we can see the encoding from the text editor itself, and can change it if we need to. How to do so obviously depends on the text editor we are using. In GNU Emacs, for example, by clicking on both the CTRL-X keys at once, then Return followed by 'f', in the last line in the window (which GNU Emacs calls a *mini-buffer*) a message will appear asking us for the new encoding and telling us what the current encoding is. In other editors we can usually access the encoding in the "Save as" menu.

Once we have checked that the encoding is correct, and have saved the file on disk, we close the editor and focus on analysing what we have written.

A look at the contents of our first source file written for ConTEXt

The first line begins with the "%" character. This is a reserved character telling ConTEXt not to process the text between that character and the end of the line where it is found. This helps when we want to write a comment on the source file that only the author can read, since it does not become part of the final document. In this example I have used this character to call attention to certain lines, explaining what it is they do.

The lines that follow begin with the "\" character, another of ConTEXt's reserved characters indicating that what follows it is the name of a command. This example shows a number of the commands found in a ConTEXt source file: the language the document is written in, the paper size, the font that will be used in the document and the way the chapters are to be formatted. Further on in other chapters we will see the details of these commands, but for the moment I am only interested in the reader seeing what they look like: they always begin with the "\", then comes the command name, and then, between curly brackets (otherwise known as braces, but we will use curly brackets in this document to make the difference clear) or square brackets, depending on the situation, the data the command needs to produce its effects. Between the name of the command and the square or curly brackets that accompany it, there may be blank spaces or line breaks.

On the 9th line of our example (I am only counting lines with some text in them) is the important `\starttext` command: it tells ConTEXt that the document's contents start from this point onwards; and, on the last line of our example, we see the command `\stoptext` that says this is where the document ends. They are two very important commands about which I will soon have more to say. Between them lies the actual contents of our document that, in our example, consists of the famous dialogue from "My Fair Lady" "The Rain in Spain...". I have written it in prose form so we can see how ConTEXt formats the paragraph.

Processing the source file

For the next step, after making sure that ConTEXt is properly installed on our system, we need to open a terminal in the same directory that our source file "rain.tex" has been saved in.

> Many text editors allow us to compile the document we have been working on without the need to open a terminal. However, the *canonical* procedure for processing a document with ConTEXt implies doing it from a terminal, by directly executing the program. I am going to do it this way (or presume that it is done this way) throughout this document for various reasons;

> the first is that I cannot know what text editor the reader is using. But the most important one is that by using a terminal, we will have access to the screen output from "`context`" and can see the messages coming from the program.

If the ConTEXt distribution that we have installed is "ConTEXt Standalone", before anything else we need to execute the *script* that tells the terminal the path and location of the files ConTEXt needs to be able to run. In Linux/Unix systems, this is done by writing the following command:

```
$> source ~/context/tex/setuptex
```

assuming we have installed ConTEXt in a directory called "`context`".

> With regard to the execution of the *script* we have just spoken about, see what it says in Appendix A in relation to the installation of "ConTEXt Standalone".

Once the variables required to run "`context`" have been loaded into memory, we can then run it. We do this by typing

```
$> context rain
```

in the terminal. Note that although the source file is called "`rain.tex`", when calling "`context`" we have omitted the file extension. Had we called the source file, for example, "`rain.mkiv`" (something I usually do so I can tell that this file was written for Mark IV), we would have had to expressly indicate the file extension by writing "`context rain.mkiv`".

After running "`context`" in the terminal, a few dozen lines will appear on the screen telling us what ConTEXt is doing. This information appears with a speed that a human being cannot follow, but we should not worry about this, since as well as being on screen, the same information is also stored in an auxiliary file, whose extension is "`.log`". This is generated at the time of processing and if necessary we can calmly consult it later.

A few seconds later, if we have written the text in our source file without making any serious errors, the terminal messages will end. The last of the messages will tell us how long it took to compile the file. A little more time is need the first time a document is compiled, since ConTEXt has to load into memory any files telling it what fonts are being used, while for further processing these are already loaded. When the final message appears telling us the time taken, the processing is complete. If everything has gone well, the directory in which we ran "`context`" will now contain three additional files:

- rain.pdf
- rain.log
- rain.tuc

The first of these is the result of our processing, or in other words it is the resulting formatted PDF. The second is the "`.log`" file storing all the information shown on screen while the file was being processed; the third is an auxiliary file that ConTEXt generates while compiling and that is used for building indexes and cross-references. For now, if everything has gone as expected, we can delete both files (`rain.log` and `rain.tuc`). If there was any problem the information in these files will help us find out where it is and will help us find a solution.

If we did not get these results, this is probably due to:

- either not having correctly installed our ConTEXt distribution, and in this case, when writing the "`context`" command in the terminal, we would have seen the message "command unknown".
- or our file was not encoded as UTF-8 and this generated a processing error.
- or perhaps the ConTEXt installed on our system was Mark II. In this version we cannot use UTF-8 encoding without expressly indicating it in the source file. We could adjust the source file so that it compiles properly but, given that this introduction refers to Mark IV, it makes no sense to continue working with Mark II: it would be best for us to install "ConTEXt Standalone".
- or we have made an error in the source file when writing a command name or the data associated with it.

  If, after running "`context`" the terminal began emitting messages, then stopped without the *prompt* reappearing, before continuing we need to press CTRL-X to abort the ConTEXt run that has been interrupted by an error.

We then need to check what has happened, and resolve it, until we get a correct compilation.

In figure 2.1 we see the contents of "`rain.pdf`". We also see that ConTEXt has numbered the page and the chapter, and has written the text in the font we indicated. There does not happen to be any hyphenation of words in this case but by default ConTEXt will hyphenate words at the end of a line in accordance with the hyphenation rules of the language chosen, and in our case the first line of our source file indicates (`\mainlanguage[en]`).

To sum up: ConTEXt has transformed the source file and generated a file where we have a document formatted according to the instructions in the source file. Any comments in that have disappeared, and as far as commands are concerned, what we have now is not their name but the results of their being executed.

2.3 The structure of our example file

In a project developed in just a single source file, the structure is very simple and marked by the commands `\starttext` ... `\stoptext`. Everything between the

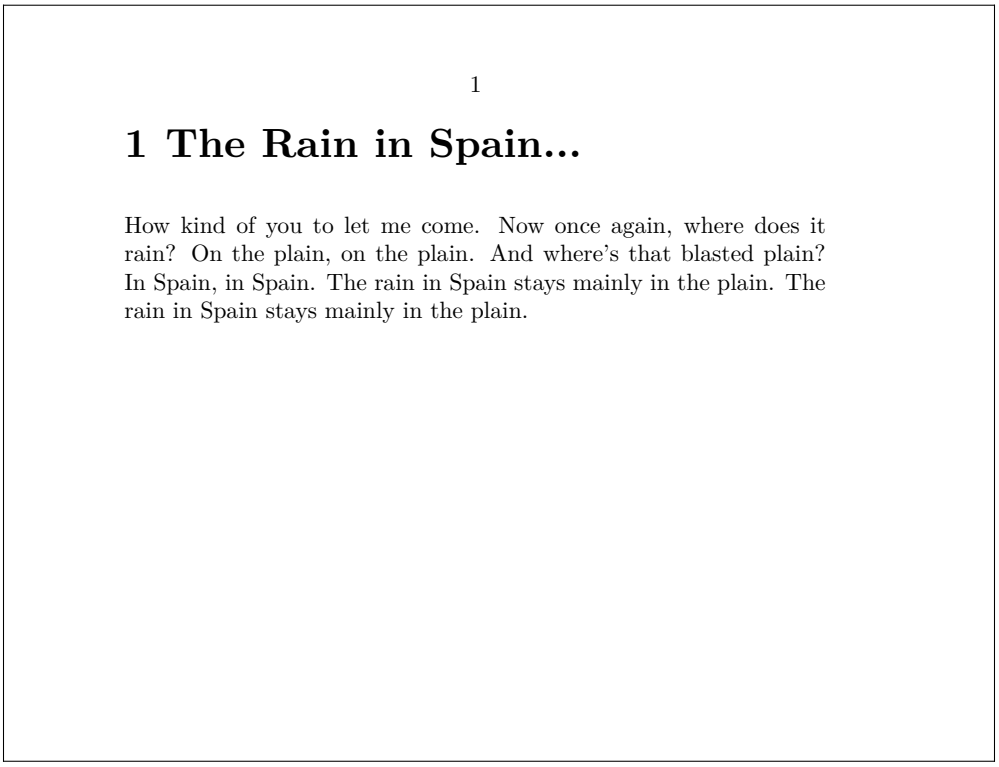


Figure 2.1 The rain in Spain...

first line of the file and the command `\starttext` is called the *preamble*. The contents of the actual document are inserted between the commands `\starttext` and `\stoptext`. In our example the preamble includes three global configuration commands: one to indicate the language of our document (`\mainlanguage`), another to indicate the size of the pages (`\setuppapersize`) which is "S5" in our case, representing the dimensions of a computer screen, and a third command (`\setuphead`) which allows us to configure what the chapter titles look like.

The body of the document is framed between the commands `\starttext` and `\stoptext`. These commands indicate the beginning and end points of the processable text respectively: between them we need to include all the text we want ConTEXt to process, along with commands that should not affect the whole document but only parts of it. For now let us assume that the commands `\starttext` and `\stoptext` are obligatory in every ConTEXt document, even though further on, when speaking about multifile projects (section 4.6) we will see that there are some exceptions to this.

2.4 Some additional details on how to run "context"

The "context" command with which we began processing our first source file earlier is really a LUA *script*, meaning a small LUA program that, after performing some checks, calls on LuaTeX, since this is what processes the source file.

We could call "context" with various options. The options are introduced immediately after the command name, preceded by two dashes. If we wish to introduce more than one option, we separate them with a space. The "help" option gives us a list of all the options, with a brief explanation of each:

```
$>context --help
```

Some of the more interesting options are as follows:

interface: As I already said in the introductory chapter, the ConTEXt interface has been translated into various languages. By default the interface is in English, however this option allows us to tell it to use Dutch (nl), French (fr), Italian (it), German (de) or Romanian (ro).

purge, purgeall: Delete the auxiliary files generated during processing.

result=Name: Indicates the name that the resulting PDF file should have. By default it will be the same as the source file being processed, with the extension .PDF.

usemodule=list: Load the modules indicated before running ConTEXt (a module is an extension of ConTEXt that is not part of its core, and that provides some additional utility).

useenvironment=list: Load the environment files indicated before running ConTEXt (an environment file is a file with configuration instructions).

version: Show the ConTEXt version.

help: print help information on program options.

noconsole: Suppress sending messages to the screen during compilation. However, these messages are still saved in the .log file.

nonstopmode: Carry out the compilation without stopping when there are errors. this does not mean that the error is not produced, but that when ConTEXt encounters an error, even one it can recover from, it will continue compiling till the end or until it encounters an error it cannot recover from.

batchmode: A combination of the two previous options. It runs without interruption and omits any screen messages.

In the early steps of learning ConTEXt I do not think it is a good idea to use the last three options since when an error is produced, we will have no clue as to where it is or what has produced it. And believe me, dear readers, sooner or later you will have an error during processing.

2.5 Managing errors

While working with ConTEXt it is inevitable that sooner or later there will be some errors during processing. We can basically group the errors into these four categories:

1. **Writing errors**. These are produced when we make a mistake with the command name. In this case we will be sending the compiler an order it does not understand. Such as when, for example, instead of writing the command `\TeX` we write `\Tex` with a final lower case 'x', given that ConTEXt differentiates between upper case and lower case and therefore sees "TeX" and "Tex" as different words; or if the functioning options of a command are placed inside square brackets instead of curly brackets, or if we try to use reserved characters as if they were normal characters, etc.

2. **Errors of omission**. In ConTEXt there are instructions that begin a task that require that we also explicitly indicate when it ends; like the reserved character $ that enables the maths mode which continues until it is disabled, and if we forget to disable it, an error is generated when a text or instruction that makes no sense in maths mode is encountered. And the same if we begin a text block with the reserved '{' character or with a `\startSomething` command and further on the explicit closing '}' or `\stopsomething` command is not found.

3. **Conception errors**. This is what I call errors produced when a command is called that requires certain arguments but they are not provided, or when the syntax that calls the command is incorrect.

4. **Situation errors**. There are some commands that are designed to work only in certain contexts or environments, and are not recognised outside of them. This happens especially in the maths mode: some ConTEXt commands only work when writing mathematical formulas and if called in another environment they generate an error.

What do we do when "`context`" warns us, while processing, that an error has been produced? The first thing, obviously, is to determine what the error is. For that we

need to analyse the "`.log`" file generated during processing; although sometimes this is not necessary, since the error is of such a kind that it has immediately forced processing to stop, in which case the error message will be visible in the same terminal where we have run "`context`".

```
3     \setuppapersize % Paper size
4       [S5]
5
6     \setupbodyfont
7       [modern,12pt] % Main font
8
9     \setuphead      % Chapter titles in bold
10       [chapter]
11       [style=\bfc]
12
13 >>  \startext  % Begin the document
14
15     \startchapter[title=The rain in Spain]
16
17     How kind of you to let me come.
18     Now once again, where does it rain?
19     On the plain, on the plain.
20     And where's that blasted plain?
21     In Spain, in Spain.
22     The rain in Spain stays mainly in the plain.
23     The rain in Spain stays mainly in the plain.

mtx-context     | fatal error: return code: 256
```

Figure 2.2 Screen output in the case of a compilation error

For example, if in our test file, "`rain.tex`", by mistake, instead of `\starttext` we had written `\startext` (with only one 't'), a very common mistake, when running "`context rain`" the processing will stop and in the terminal we can see the information shown in figure 2.2. There we can see that the lines of our source file are numbered, and in one of them, in this case number 13, between the number and the line of text, the compiler has added "`>>`" to indicate that this is the line where it has found an error. The file "`rain.log`" will give us more clues. In our example it is not such a big file, since the source being compiled is much reduced; in other cases it might contain an overwhelming amount of information. But we must dive into it. If we open "`rain.log`" with a text editor we will see that it has stored everything that ConTEXt is doing. We need to find a line there that begins with an error warning and for this we can use the text editor's search function. We will be looking for "tex error", and that will bring us to the following lines:

```
tex error       > tex error on line 13 in file |
                  /home/jq/context/docs/rain.tex: ! Undefined control sequence

l.13 \startext
              % Begin the document
```

Note: The first line telling us about the error in the "`rain.log`" file is very long. To make it look good, bearing in mind the width of the page, I have split it in two. The character '|' shows the point where I have split it.

If we pay attention to the three lines of the error message, we see that in the first it tells us what line number has produced the error (line 13) and what kind of error it is: "Undefined control sequence", or, which is the same thing: unknown control sequence, in other words, unknown command. The two following lines of the log file show us line 13, split at the point that produced the error. So there is no doubt that the error lies in `\startext`. We read it carefully and with luck and experience, we will realise that we have written "startext" and not "starttext" (with a double 't').

Think of the fact that computers are very good and very fast at carrying out instructions, but very slow at reading our mind, and the word "startext" is not the same as "starttext". The program knows how to execute the latter, not the former. It does not know what to do with that.

At other times, finding the error will not be so easy. Especially when the error consists of the fact that something has begun but where it must end has not been indicated. At times, instead of searching for "tex error" in the "`.log`" file, we should be looking for an asterisk. This character at the beginning of a line in the file is not so much a fatal error as a warning. However, warnings can be helpful for finding the error.

And if the information in the "`.log`" file is not enough, we would need to go through our main file, bit by bit, looking for the error. A good strategy for this is to change the location of the `\stoptext` command. Remember that ConTEXt stops processing the text when it finds this command. Therefore, if I place a `\stoptext` more or less halfway through the file and compile it, only that first half will be compiled; if the error happens again then I know it is in the first half of the source file, and if not, then it means it is in the second half... and so on, bit by bit, changing the location of the `\stoptext` command, we will be able to find where the error is. Once we have found it, we can then try to understand and correct it or, if we cannot understand why the error has been produced, at least, by finding where it is, we can try writing things in another way to avoid reproducing it. This latter solution, of course, can only apply if we are the author. If we simply typeset a text for someone else, we cannot alter it and will have to keep investigating until we discover the reasons for the error and its possible solution.

In practice, when a relatively long document is produced with ConTEXt it is usually compiled from time to time as the document is being drafted, so that if it throws an error we will be more or less clear about the new part since the last time we processed the file, and why it has thrown an error.

Chapter 3
Commands and other fundamental concepts of ConTEXt

We have already seen that in the source file, as well as in the actual contents of our future formatted document, we find the instructions needed to explain to ConTEXt how we want our manuscript to be transformed. These instructions can be called "commands", "macros" or "control sequences".

From the point of view of ConTEXt's internal functioning (actually, TEX's functioning), there is a difference between *primitives* and *macros*. A primitive is a simple instruction that cannot be broken down into other simpler instructions. A macro is an instruction that can be broken down into other simpler instructions which, in turn, can also perhaps be broken down into still others, and so on and so on. Most of ConTEXt's instructions are, in fact, macros. From the programmer's perspective, the difference between macros and primitives is important. But from the user's perspective the issue is not so important: in both cases what we have are instructions that are carried out without our need to worry about how they function at a low level. Therefore, ConTEXt documentation commonly talks about a *command* when it takes the user's perspective, and a *macro* when it takes the programmer's perspective. Since we are only taking the user's perspective in this introduction, I will use either term, regarding them as synonymous.

Commands are orders given to ConTEXt to do something; we *control* the program's performance through them. Thus KNUTH, the father of TEX, uses the term *control sequences* to refer to

both primitives and macros, and I think this is the most accurate term of them all. I will use it when I believe it is important to distinguish between *control symbols* and *control words*.

ConTEXt's instructions are basically of two kinds: reserved characters, and commands properly so called.

3.1 ConTEXt's reserved characters

When ConTEXt is reading the source file made up only of text characters, since it is a text file it needs to somehow distinguish what is actual text to be formatted, and what are the instructions it has to carry out. ConTEXt's reserved characters are what enable it to make this distinction. In principle, ConTEXt will assume that every character in the source file is text to be processed, unless it is one of the 11 reserved characters which are to be treated as an *instruction*.

Only 11 instructions? No. There are only 11 reserved characters; but since one of them, the"\" character, has the function of converting the character or characters immediately following it into an instruction, then really the potential number of commands is unlimited. ConTEXt has around 3000 commands (adding up the commands exclusive to Mark II, Mark IV and the ones common to both versions).

The reserved characters are as follows:

\ % { } # ~ | $ _ ^ &

ConTEXt interprets them in the following way:

\ This character is the most important of all for us: it indicates that what comes immediately after must not be interpreted as text but as an instruction. It is called the "Escape character" or "Escape sequence" (even though it has nothing to do with the "Esc" key found on most keyboards).[1]

% Tells ConTEXt that what follows up to the end of the line is a comment that must not be processed or included in the final formatted file. Introducing comments into the source file is extremely useful. A comment can help explain why something has been done in a certain way, and this is very helpful in completed source files, in view of later revision when sometimes

[1] In computer terminology, the key that affects the interpretation of the following character is called the *escape character*. By contrast, the *escape key* on keyboards is called this because it generates character 27 in ASCII code, which is used as the escape character in this encoding. Today, the uses of the Escape key are more associated with the idea of cancelling an ongoing action.

we cannot remember why we did what we did; or it can also help as a reminder to ourselves about something we might need to revise. It can even be used to help locate the cause of a certain error in the source file, since by placing a comment mark at the beginning of a line, we exclude that line from being compiled, and can see if it was that line that was causing the error; it can also be used to store two different versions of the same macro, and that way get different results after compiling; or to prevent a snippet from being compiled that we are not sure about but without deleting it from the source file in case we want to return to it later ... etc. Once we have opened up the possibility that our source file contains text that nobody but ourselves should see, our uses of this character are only limited by our own imagination. I admit that this is one of the utilities I miss most when the only remedy for writing a text is a word processor.

`{` This character opens a group. Groups are blocks of text affected by certain features. We will talk about them in section 3.8.1.

`}` This character closes a group previously opened with `{`.

`#` This character is used for defining macros. It refers to the macro's arguments. See section 3.7.1 in this chapter.

`~` Introduces a white space into the document to prevent a line break, meaning that two words separated by the `~` character will always remain on the same line. We will speak about this instruction and where it should be used in section 11.3.1.

`|` This character is used to indicate that two words joined by a separating element constitute a compound word that can be divided by syllables into the first component, but not into the second component. See section 10.4.

`$` This character is a switch for the maths mode. It enables that mode if it wasn't enabled, or disables it if it was. When in maths mode, ConTEXt applies some fonts and rules that differ from normal ones, aimed at optimising the writing of mathematical formulas. Even though writing mathematics is a very important use of ConTEXt, I will not develop this in this introduction. Being a literary man, I don't feel up to it!

`_` This character is used in maths mode to indicate that what follows is a subscript. So, for example, to get x_1, we need to write `$x_1$`.

`^` This character is used in maths mode to indicate that what follows is a superscript. So for example, to get $(x+i)^{n^3}$ we need to write `$(x+i)^{n^3}$`.

`&` It says in the ConTEXt documentation that this is a reserved character, but it does not say why. In Plain TEX this character basically has two uses: it is

used to align columns in basic table environments, and, in a maths context, so that what follows is to be treated as normal text. In the introductory manual "ConTEXt Mark IV, an Excursion", although it does not say what it is for, there are examples of its use in mathematical formulas, though not of the kind it had in Plain TEX, but to align columns within complex functions. As I am a literary person, I do not feel I can carry out further tests to see what the precise use of this reserved character is for.

It can be assumed that in selecting which characters would be reserved ones, they would be characters available on most keyboards but ones not usually used in written scripts. However, although not so common, there is always the possibility that some of them will figure in our documents, like for example, when we want to write that something costs a 100 dollars ($100), or that in Spain, the percentage of drivers over 65 years of age was 16% in 2018. In these cases we must not write the reserved character directly but use a *command* that will output the reserved character properly in the final document. The command for each of the reserved characters is found in table 3.1.

Reserved character	Command that generates it
\	\backslash
%	\%
{	\{
}	\}
#	\#
~	\lettertilde
|	\\|
$	\$
_	_
^	\letterhat
&	\&

Table 3.1 Writing reserved characters

Another way of getting the reserved characters is with the \type command. This command sends what it takes as an argument to the final document without processing it in any way, and therefore without interpreting it. In the final document, the text received from \type will be shown in the monospaced font typical of computer terminals and typewriters.

Normally we would enclose the text that \type has to show between curly brackets. However, when this text itself includes opening or closing curly brackets, instead of them we must enclose the text between two equal characters that are not part of the text that constitutes the argument of \type. For example: \type*{*, or \type+}+.

If, by mistake, we use one of the reserved characters directly, other than for the purpose which it is intended, because we have forgotten that it is a reserved character and cannot be used like a normal one, then three things can happen:

1. Most commonly, an error is generated when compiling.

2. We get an unexpected result. This happens especially with "~" and "%"; in the former case, instead of the "~" we expected in the final document, a white space will be inserted; and in the latter case, everything on the same line will stop being processed, starting from "%". Improper use of the "\" too can produce an unexpected result if it or the characters immediately after it make up a command that ConTEXt knows about. However, more commonly when we incorrectly use the "\" we will have a compiling error.

3. No problem occurs: This happens with three of the reserved characters used mainly in mathematics (_ ^ &): if used outside of this environment they are treated as normal characters.

Point 3 is my conclusion. The truth is that I not found anywhere in the ConTEXt documentation that tells us where these reserved characters can be used directly; in my tests, however, I have not seen any error when this is done; unlike, for example, in LATEX.

3.2 Commands themselves

Commands themselves always begin with the "\" character. Depending on what comes immediately after the escape sequence, a distinction is made between:

a. **Control symbols.** A control symbol begins with the escape sequence ("\") and consists exclusively of a character other than a letter, as for example "`\,`", "`\1`", "`\'`" or "`\%`". Any character or symbol that is not a letter in the strict sense of the term can be a control symbol, including numbers, punctuation marks, symbols and even a blank space. In this document, to represent a blank space (white space) when its presence needs to be highlighted, the symbol I use is ␣. In fact, "\␣" (a backslash followed by a blank space) is a commonly used control symbol, as we will soon be able to see.

A blank or white space is an "invisible" character, which is a problem in a document like this, where at times we need to clearly specify what needs to be written in a source file. Knuth was already aware of the problem, and in his "The TEXBook" he began the custom of representing significant blank spaces with the "␣" symbol. So, for example, if we wanted to show that two words in the source file need to be separated by two blank spaces, then we would write "word1␣␣word2".

b. **Control words.** If the character immediately following the backslash is a letter properly speaking, the command will be a *Control word*. this group of

commands is the most numerous and its feature is that the command name can only consist of letters; numbers, punctuation marks or any other kind of symbol are not allowed. Only lower case or upper case letters. Bear in mind, on the other hand, that ConTEXt makes a distinction between lower case and upper case, meaning that the `\mycommand` and `\MyCommand` commands are different. But `\MyCommand1` and `\MyCommand2` would be considered the same, since not being letters, '1' and '2' are not part of the command names.

The ConTEXt reference manual contains no rules on command names, nor do the rest of the "manuals" included with "ConTEXt Standalone". What I stated in the previous paragraph is my conclusion based on what happens in TEX (where, besides, characters like accented vowels that do not appear in the English alphabet are not thought of as "letters"). This rule makes it possible to offer a good explanation for the absorption of white space after a command name.

When ConTEXt is reading a source file and finds an escape character ("\"), it knows that a command will follow. It then reads the first character following the escape sequence. If it is not a letter, it means the command is a control symbol and consists only of this first symbol. But on the other hand, if the first character after the escape sequence is a letter, then ConTEXt will continue to read each character until it finds the first non-letter, and then it knows that the command name has finished. This is why command names that are control words cannot contain characters that are not letters.

When the "non-letter" at the end of the command name is a blank space, it is assumed that the blank space is not part of the text to be processed, but was inserted exclusively to indicate where the command name ended, so ConTEXt gets rid of this space. This produces an effect that surprises ConTEXt beginners, because when the effect of the command in question implies writing something in the final document, the written output of the command is connected to the next word. For example, the following two sentences in the source file

```
Knowing \TeX helps with learning \ConTeXt.
Knowing \TeX, although not essential, helps with learning \ConTeXt
```

produce the following results respectively:

Knowing TEXhelps with learning ConTEXt.
Knowing TEX, although not essential, helps with learning ConTEXt.[1]

Note how, in the first case, the word "TEX" is connected to the word that follows but not in the second case. This is because, in the first case in the source file, the

[1] **Note:** two conventions are followed in cases where, to illustrate something in this introduction, a fragment of source code is written as well as the result of compiling it: sometimes the code and the result of its compilation are placed next to each other in a two-column paragraph; other times the code is written in dark magenta shade which is generally used in this document to represent ConTEXt commands, and the result of its compilation in red.

first “non-letter” after the command name \TeX was a blank space, suppressed because ConTEXt assumed it was there only to indicate the end of a command name, while in the second instance there was a comma, and since this is not a blank space, it has not been suppressed.

On the other hand, this problem is not solved simply by adding an extra blank space, and writing, for example,

```
Knowing \TeX␣␣helps with learning \ConTeXt[1].
```

will not solve the problem, because a ConTEXt rule (that we will see in section 4.2.1) is that a blank space absorbs all the blanks and tabs that follow it. Therefore, when we have this problem (which fortunately does not happen too often) we must make sure that the first “non-letter” after the command name is not a blank space. There are two candidates for this:

- The reserved characters “{}”. The reserved character “{”, as I have said, opens a group, and “}” closes a group, therefore the sequence “{}” introduces an empty group. An empty group has no effect on the final document, but it helps ConTEXt to know that the command name prior to it has finished. Or we could also create a group around the command in question, for example by writing “{\TeX}”. In either case, the result will be that the first “non-letter” after \TeX is not a blank space.
- The control symbol “\␣” (a backslash followed by a blank space, see the note on page 46). The effect of this control symbol is to insert a blank space in the final document. To understand ConTEXt's logic properly, it may be worth taking some time to see what happens when ConTEXt encounters a control word (for example \TeX) followed by a control symbol (e.g. “\␣”):
  - ConTEXt encounters the \ character followed by a ‘T’ and knowing that this comes before a control word, it keeps reading characters until it comes to a “non-letter”, something that happens when it comes to the \ character introducing the next control symbol.
  - Once it knows that the command name is \TeX, it runs the command and prints TEX in the final document. It then returns to the point where it stopped reading to check the character immediately after the second backslash.
  - It checks that it is a blank space, meaning a “non-letter” which means that the control sequence is exactly that, so it can run it. It does so, and inserts a blank space.
  - Finally, it returns once more to the point where it stopped reading (the blank space that was the control symbol) and continues to process the source file from there onwards.

[1] Regarding the “␣” symbol, recall the note on page 46.

I have explained this mechanism in some detail, as the elimination of blank spaces often surprises newcomers. However, it should be noted that the problem is relatively minor, as the control words do not usually print directly to the final document, but affect the format and appearance. By contrast, it is quite common for control symbols to print something to the final document.

> There is a third procedure to avoid the problem of blank space, which consists in defining (TEX style) a similar command and including a "non-letter" at the end of the command name. For example, the following sequence:
>
> ```
> \def\txt-{\TeX}
> ```
>
> would create a command called \txt, that would do exactly the same as the command \TeX and only function correctly if called with a hyphen after it \txt-. This hyphen is not technically part of the command name, but it will not work unless the name is followed by a hyphen. Why this is so has to do with the mechanism for defining TEX macros, and it is too complex to explain here. But it works: once this command is defined, every time we use \txt-, ConTEXt substitutes it with \TeX by eliminating the hyphen, but using it internally to know that the command name is already finished, so a blank space immediately after it would not be deleted.
>
> This 'trick' will not work correctly with the \define command, which is a specifically ConTEXt command for defining macros.

3.3 Scope of the commands

3.3.1 Commands that do or do not require a scope to be indicated

Many of the ConTEXt commands, especially those that affect formatting features of fonts (bold, italic, small caps, etc.), enable a certain feature that remains enabled until another command is encountered that disables it, or that enables another feature incompatible with it. For example, the command \bf enables bold, and this will remain active until it finds an *incompatible* command like, for example, \tf, or \it.

These kinds of commands do not need to take any argument, as they are not designed to apply only to certain text. It is as if they are limited to *turning on* whatever function (bold, italic, sans serif, a certain font size, etc.).

When these commands are executed within a *group* (see section 3.8.1), they also lose their effectiveness when the group they are executed in is closed. Therefore, often in order to make these commands affect only a portion of text, what is done is to generate a group containing that command and the text we want it to affect. A group is created by enclosing it between curly brackets. Therefore, the following text

```
In {\it The \TeX Book}, {\sc Knuth}
explained everything you need to know
about \TeX.
```

In *The TEXBook*, KNUTH explained everything you need to know about TEX.

creates two groups, one to determine the scope of the `\it` (italics) command and the other to determine the scope of the `\sc` (small caps) command.

By contrast with this kind of command, there are others that, because of the effect they produce or for other reasons, require an express indication of what text they are to be applied to. In these cases the text to be affected by the command is enclosed within curly brackets *immediately after the command*. As an example of this we could mention `\framed`: this command draws a frame around the text it takes as an argument, such that

```
\framed{Tweedledum and Tweedledee}
```

will produce

Tweedledum and Tweedledee

Note that although in the first group of commands (those that require an argument) curly brackets are also sometimes used to determine the field of action, this is not necessary for the command to work. The command is designed to be applied from the point where it appears. So, when determining its field of application by using brackets, the command is placed *within these brackets*, unlike in the second group of commands, where the brackets framing the text the command is to be applied to, come *after* the command.

In the case of the `\framed` command, it is obvious that the effect it produces requires an argument – the text to which it is to be applied. In other cases, it depends on the programmer whether the command is of one type or the other. So, for example, what the `\it` and `\color` commands do is quite similar: they apply a feature (format or colour) to the text. But the decision was made to program the first one without an argument, and the second as a command with an argument.

3.3.2 Commands requiring an express indication of where they begin and end (environments)

There are certain commands that determine their scope by indicating precisely the point at which they begin to be applied and the point where they cease to do so. These commands, therefore, come in pairs: one indicating when the command is to be enabled, and the other when this action must cease. “start”, followed by the command name, is used to indicate the beginning of the action, and “stop”, also followed by the command name, to indicate the end. So for example, the command

"`itemize`" becomes `\startitemize` to indicate the beginning of *itemization* and `\stopitemize` to indicate where it ends.

There is no special name for these command pairings in the official ConTEXt documentation. The reference manual and the introduction simply call them "start ... stop". Sometimes they are called *environments*, which is the name LATEX gives to a similar kind of construction, although this has the disadvantage that in ConTEXt the term "environment" is used for something else (a special kind of file that we will see when talking about multifile projects in section 4.6). Even so, since the term environment is clear, and the context will make it easy to distinguish if we are talking about *environment commands* or *environment files*, I will use this term.

Environments, therefore, consist of a command that opens or begins them, and another that closes or ends them. If the source file contains a command to open the environment that is not later closed, an error will normally be generated.[1] On the other hand, these kinds of errors are harder to find, as the error can occur a long way past where the opening command occurs. Sometimes the "`.log`" file will show us the line where the incorrectly closed environment begins; but at other times, the lack of closure of the environment means that ConTEXt misinterprets a certain passage and not in that faulty environment, meaning that the "`.log`" file is not much help to us for finding where the problem lies.

Environments can be nested, meaning another environment can be opened within an existing environment, although in the case where there are nested environments, an environment needs to be closed inside the environment it was opened in. In other words, the order in which environments are closed has to be consistent with the order in which they were opened. I believe this should be clear from the following example:

```
\startSomething
  ...
  \startSomethingElse
    ...
    \startAnotherSomethingElse
      ...
    \stopAnotherSomethingElse
  \stopSomethingElse
\stopSomething
```

In the example you can see how the "`AnotherSomethingElse`" environment has been opened inside the "`SomethingElse`" environment and needs to be closed inside it as well. To do otherwise would generate an error when compiling the file.

[1] Though not always; it depends on the environment in question and on the situation in the rest of the document. ConTEXt differs from LATEX in this regard, which is much stricter.

In general, commands designed as *environments* are ones that implement some change intended to be applied to units of text no smaller than the paragraph. For example, the "`narrower`" environment that changes the margins only makes sense when applied at paragraph level; or the "`framedtext`" environment that frames one or more paragraphs. This latter environment may help us understand why some commands are designed as environments and others as individual commands: if we wish to frame one or more words, all on the same line, we would use the command `\framed`, but if what we want framed is a whole paragraph (or several paragraphs) then we would use the "`framedtext`" environment.

On the other hand, text located within a particular environment normally constitutes a *group* (see section 3.8.1), which means that if an activation command is found inside an environment, of those commands that apply to all the text that follows, this command will apply only until the end of the environment in which it is found; and, in fact ConTEXt has an unnamed *environment* beginning with the `\start` command (no other text follows; just *start*. This is why I call it an *unnamed environment*) and finishing with the `\stop` command. I suspect that the only function this has is to create a group.

I have not read anywhere in ConTEXt documentation that one of the effects of environments is to group their contents, but this is the result of my tests with a number of the predefined environments, though I must admit that my tests have not been too exhaustive. I have simply checked some environments chosen at random. My tests show, however, that such a statement, if true, would only be so for some predefined environments: those created with the `\definestartstop` command (explained in the section 3.7.2) do not create any group, unless when defining the new environment we include the commands needed to create the group (see section 3.8.1).

It is also my assumption that the environment I have called the *unnamed* (`\start`) environment is only there to create a group: it does create a group, but whether or not it has some other use I do not know. This is one of the undocumented commands in the reference manual.

3.4 Command operation options

3.4.1 Commands that can work in several different ways

Many commands can work in more than one way. In such cases there is always a predetermined way of working that can be altered by indicating the parameters corresponding to the desired operation in brackets after the command name.

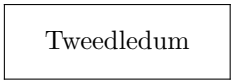
We find a good example of what I have just said with the `\framed` command mentioned in the previous section. This command draws a frame around the text it takes as an argument. By default, the frame has the height and width of the

text it is applied to; but we can indicate a different height and width. Thus we can see the difference between how the default `\framed` functions:

```
\framed{Tweedledum}
```

and how a customised version functions:

```
\framed
  [width=3cm, height=1cm]
  {Tweedledum}
```



In the second example, between square brackets we have indicated a specific width and height for the frame that surrounds the text it takes as an argument. Within the brackets, the different configuration options are separated by a comma; blank spaces and even line breaks (as long as they are not a double line break) between two or more options, are not taken into consideration so that, for example, the next four versions of the same command produce exactly the same result:

```
\framed[width=3cm,height=1cm]{Tweedledum}

\framed[width=3cm,    height=1cm]{Tweedledum}

\framed
  [width=3cm, height=1cm]
  {Tweedledum}

\framed
  [width=3cm,
    height=1cm]
  {Tweedledum}
```

It is obvious that the final version is the easiest to read: we can see at first sight how many options there are and how they are used. In an example like this with only two options, perhaps it might not seem so important; but in cases where there is a long list of options, if each of them has its own line in the source file it makes it easier to *understand* what the source file is asking ConTEXt to do, and also, if necessary, to discover a potential error. Therefore, this last format (or similar) for writing commands is the 'preferred' one for users.

As for the syntax of configuration options, see further ahead in (section 3.5).

3.4.2 Commands that configure how other commands work (\setupSomething)

We have already seen that commands that support various possibilities in how they function always have a default way of working. If one of these commands is called several times in our source file, and we would like to alter the default for them all, rather than changing these options each time the command is called, it is much more convenient and efficient to change the default. To do this there is almost always a command available whose name begins with `\setup`, followed by the name of the command whose default options we wish to change.

The `\framed` command we have been using as an example in this section continues to be a good example. So, if we are using a lot of frames in our document, but they all need precise measurements, then it would be best to reconfigure how `\framed` works, doing so with `\setupframed`. Thus

```
\setupframed
  [
    width=3cm,
    height=1cm
  ]
```

will ensure that from then on, every time we call `\framed`, by default it will then generate a frame 3 centimetres wide by 1 centimetre high, without needing to indicate this expressly each time.

There are some 300 commands in ConTEXt that allow us to configure how other commands function. Thus, we can configure the default functioning of frames (`\framed`), lists ("`itemize`"), chapter titles (`\chapter`), or section titles (`\section`), etc.

3.4.3 Setting up customised versions of configurable commands (\defineSomething)

Continuing with the `\framed` example, it is obvious that if our document uses several kinds of frames, each with different measurements, the ideal would be that we could *predefine* different configurations of `\framed`, and associate them with a particular name so we could use one or other of them as needed. We can do this in ConTEXt with the `\defineframed` command, whose syntax is:

```
\defineframed[Name][Configuration]
```

where *Name* is the name assigned to the particular kind of frame to be configured; and *Configuration* is the particular configuration associated with this name.

The effect of all this will be that the indicated configuration is associated with the name we have established, which, to all intents and purposes, will work as if it

were a new command, and we can use this in any context where we would have been able to use the original command (`\framed`).

This possibility does not only exist for the concrete case of the `\framed` command, but for many of the commands that have a `\setup` possibility. The combination of `\defineSomething` + `\setupSomething` is a mechanism that gives ConTEXt its extreme power and flexibility. If we make a detailed examination of what the `\defineSomething` command does, we see that:

- First of all, it clones a particular command that supports a variety of configurations.
- It associates this clone with the name of a new command.
- Finally, it sets a predetermined configuration for the clone, different from how the original command was configured.

In the example we have given, we were configuring our special frame at the same time as we were creating it. But we can also create it first and configure it later, because, as I said, once the clone is created it can be used where the original could have been used. So for example, if we have created a frame called "`MySpecialFrame`", we can configure it with `\setupframed` indicating the actual frame we want to configure. In this case the `\setup` command will take a new argument with the name of the frame to be configured:

```
\defineframed[MySpecialFrame]

\setupframed
  [MySpecialFrame]
  [ ... ]
```

3.5 Summary of command syntax and options, and on the use of square and curly brackets when calling them

Summing up what we have seen so far, we see that in ConTEXt

- Commands properly so called always begin with the "\" character.
- Some commands can take one or several arguments.
- Arguments that tell the command *how* it must function or that affect how it works in some way, are introduced inside square brackets.
- Arguments that tell the command what part of the text it must act on are introduced inside curly brackets.

When the command will only act on one letter, as is the case, for example with the \buildtextcedilla command (just to give an example – the 'ç' so often used in Catalan), the curly brackets around the argument can be omitted: the command will apply to the first character that is not a blank space.

- Some arguments can be optional, in which case we can omit them. But what we can never do is to change the order of the arguments the command is expecting.

Arguments introduced between square brackets can be of various kinds. Mainly:

- They can take only a single value which will almost always be one word or a phrase.
- They can take various options, in which case they can:
  - Be represented by just one word that could be a symbolic name (one that ConTEXt knows the meaning of), a measure or dimension, a number, the name of another command, etc.
  - Consist of names of variables that must be given a value. In this case the official definition of the command (see section 3.6) always tells us what kind of value each of the options expects.
    - ⋆ When the value the option expects is text, this can contain blank spaces and also commands. In these cases it is sometimes convenient to enclose the value of the option between curly brackets.
    - ⋆ When the value an option expects is a command, normally we can indicate more than one command as the value of an option, although sometimes we need to enclose all the commands assigned to the option between curly brackets. We must also enclose the contents of the option between curly brackets if any of the commands included in it takes an option between square brackets.

  In both cases the different options that are to take the same argument will be separated by commas. White space and line breaks (other than doubles) between the different options are ignored. White space and line breaks between the different arguments to a command are also ignored.
- Finally, it is never the case with ConTEXt that the same argument simultaneously takes options consisting of a word and options consisting of a variable that must be explicitly assigned a value. In other words, we can have options like

  ```
  \command[Option1, Option2, ...]
  ```

  and others like

```
\command[Variable1=value, Variable2=value, ...]
```

But we can never find a mixture of both:

```
\command[Option1, Variable1=value, ...]
```

3.6 The official list of ConTEXt commands

Amongst the ConTEXt documentation, there is an especially important document with a list of all the commands, indicating for each of them how many arguments they expect and of what kind, as well as the different options envisaged and their permitted value. This document is called "`setup-en.pdf`", and is generated automatically for each new version of ConTEXt. It can be found in the directory called "`tex/texmf-context/doc/context/documents/general/qrcs`".

> In fact, the "`qrc`" has seven versions of this document, one for each of the languages that has a ConTEXt interface: German, Czech, French, Dutch, English, Italian and Romanian. For each of these languages there are two documents in the directory: one called "`setup-LangCode`" (where LangCode is the two international language identification letter-code) and a second document called "`setup-mapping-LangCode`". This second document contains a list of commands in alphabetical order and indicates the command *prototype*, but without further information on the likely values for each argument.

This document is fundamental for learning to use ConTEXt, because it is there that we can find out if a certain command exists or not; this is especially useful, bearing in mind the COMMAND (or ENVIRONMENT) + setupCOMMAND + defineCOMMAND combination. For example, if I know that a blank line is introduced with the `\blank` command, I can find out if there is a command called `\setupblank` that lets me configure it, and another that allows me to set up a customised configuration for blank lines, (`\defineblank`).

> "`setup-en.pdf`", therefore, is fundamental for learning ConTEXt. But I would really prefer, first of all, for it to tell us if a command only works in Mark II or Mark IV, and especially, that if instead of only telling us about the number of type of arguments each command allows, it would tell us what these arguments are for. This would greatly reduce the shortcomings of the ConTEXt documentation. There are some commands that allow optional arguments that I don't even mention in this introduction because I don't know what they are for and, since they are optional, nor is there any need to mention them. This is extremely frustrating.

3.7 Defining new commands

3.7.1 General mechanism for defining new commands

We have just seen how, with \defineSomething we can clone a pre-existing command and develop a new version of it from there, that will function, to all intents and purposes, as a new command.

Along with that possibility, which is only available to some specific commands (quite a few, certainly, but not all), ConTEXt has a general mechanism for defining new commands that is extremely powerful though, in some of its uses, also quite complex. In a text like this one, aimed at beginners, I think it best to introduce it by starting with some of its simplest uses. The simplest of all is to associate snippets of text with a word, so that each time this word appears in the source file, it is replaced by the text linked to it. This will allow us, on the one hand, to save a lot of typing time and, on the other hand, as an extra advantage, it reduces the possibilities of making mistakes when typing, while ensuring that the text in question is always written the same way.

Let us imagine, for example, that we are writing a treatise on alliteration in Latin texts, where we are often quoting the Latin sentence "*O Tite tute Tati tibi tanta tyranne tulisti*" (O Titus Tatius, you tyrant, so much you have brought upon yourself!). It is a fairly long sentence in which two of the words are proper names and start with capital letters, and where, let us admit it, as much as we might love Latin poetry, it is easy for us to "trip up" when writing it down. In this case, we could simply put in the preamble of our source file:

```
\define\Tite{\quotation{O Tite tute Tati tibi tanta tyranne tulisti}}
```

Based on such a definition, each time the command \Tite appears in our source file, it will be substituted by the sentence indicated, and it will also be between quotation marks just as the original definition had it, which allows us to ensure that the way that sentence appears will always be the same. We could also have written it in italics, with a larger font size... whatever we want. The important thing is that we only have to write it once, and throughout the text it will be reproduced exactly as it was written, as often as we want. We could also create two versions of the command, called \Tite and \tite, depending on whether the sentence needs to be written using capital letters or not. The replacement text can be pure text, or include commands, or form mathematical expressions in which there is more chance of mistyping (at least for me). For example, if the expression $(x_1, ..., x_n)$ needs to appear regularly in our text, we could create a command to represent it. For example

```
\define\xvec{$(x_1,\ldots,x_n)$}
```

so that whenever \xvec appears in the text, it is replaced by the expression associated with it.

The general syntax of the \define command is as follows:

```
\define[NumArguments]\CommandName{TeXtToReplace}
```

where

- **NumArguments** refers to the number of arguments the new command will take. If it doesn't need to take any, as in the examples given so far, this would be omitted.
- **CommandName** refers to the name the new command will have. Here, the general rules regarding command names apply. The name could be a single character that is not a letter, or one or more letters without including any "non-letter" character.
- **TextToReplace** contains the text that will replace the name of the new command, each time it is found in the source file.

The possibility of providing the new commands with arguments in their definition gives this mechanism great flexibility, as it allows a variable replacement text to be defined according to the arguments taken.

For example: let us imagine that we want to write a command that produces the opening of a business letter. A very simple version of this would be:

```
\define\LetterHeading{
  \rightaligned{Peter Smith}\par
  \rightaligned{Consultant}\par
  Maryborough, \date\par
  Dear Sir,\par
  }
```

but it would be preferable to have a version of the command that would write the name of the recipient in the header. This would require the use of a parameter that communicates the name of the recipient to the new command. This would require redefining the command as follows:

```
\define[1]\LetterHeading{
  \rightaligned{Peter Smith}\par
  \rightaligned{Consultant}\par
  Maryborough, \date\par
  Dear Mr #1,\par
  }
```

Note that we have introduced two changes in the definition. First of all, between the key word \define and new name for the command, we have included a 1 between square brackets ([1]). This tells ConTEXt that the command we are defining will take one argument. Further on, in the last line of the command definition, we have

written "`Dear Mr #1,`", using the reserved character "`#`". This indicates that at the point in the replacement text where "`#1`" appears, the contents of the first argument will be inserted. If it had two parameters, "`#1`" would refer to the first parameter and "`#2`" to the second. In order to call the command (in the source file) after the command name, the arguments must be included between curly brackets, each argument with its own set. So, the command that we have just defined should be called in the following way in the text of our source file:

```
\LetterHeading{Name of addressee}
```

For example: `\LetterHeading{Anthony Moore}`.

We could still further improve the previous function, because it assumes that the letter will be sent to a man (it puts "Dear Sir"), so perhaps we could include another parameter to distinguish between male and female addressees. for example:

```
\define[2]\LetterHeading{
  \rightaligned{Peter Smith}\par
  \rightaligned{Consultant}\par
  Maryborough, \date\par
  #1\ #2,\par
  }
```

so that the function would be called, for example, with

```
\LetterHeading{Dear Ms}{Eloise Merriweather}
```

although this is not very elegant (from a programming point of view). It would be preferable for symbolic values to be defined for the first argument (man/woman; 0/1; m/f) so that the macro itself would choose the appropriate text according to this value. But explaining how to achieve this requires us to get into more depth than I think the novice reader can understand at this stage.

3.7.2 Creating new environments

To create a new environment, ConTEXt provides the `\definestartstop` command whose syntax is as follows:

```
\definestartstop[Name][Options]
```

In the *official* definition of `\definestartstop` (see section 3.6) there is an additional argument that I have not put above because it is optional, and I have not been able to found out what it is for. Neither the introductory ConTEXt "Excursion", nor the incomplete reference manual explain it. I had assumed that this argument (which must be entered between the name and the configuration) could be the name of some existing environment that would serve as an initial model for the new environment, but my tests show that this assumption was wrong. I have looked on the ConTEXt mailing list and I have not seen any use of this possible argument.

where

- **Name** is the name the new environment will have.
- **Configuration** allows us to configure the behaviour of the new environment. We have the following values with which we can configure it:
  - `before` – Commands that have to be run before entering the environment.
  - `after` – Commands that have to be run after leaving the environment.
  - `style` – Style that the text of the new environment must have.
  - `setups` – Set of commands created with `\startsetups ... \stopsetups`. This command and its use is not explained in this introduction.
  - `color, inbetween, left, right` – Undocumented options that I have not been able to make work. We can assume what some do because of their name, for example `color`, but from more tests I have done, indicating some value for that option, I do not see any change within the environment.

An example of the definition of an environment could be as follows:

```
\definestartstop
  [TextWithBar]
  [before=\startmarginrule\noindentation,
    after=\stopmarginrule,
    style=\ss\sl
  ]

\starttext

The first two fundamental laws of human stupidity state unambiguously
that:

\startTextWithBar

  \startitemize[n,broad]

    \item Always and inevitably we underestimate the number of stupid
    individuals in the world.

    \item The probability that a given person is stupid is independent
    of any other characteristic of the same person.

  \stopitemize

\stopTextWithBar

\stoptext
```

The result would be:

> The first two fundamental laws of human stupidity state unambiguously that:
>
> 1. *Always and inevitably we underestimate the number of stupid individuals in the world.*
> 2. *The probability that a given person is stupid is independent of any other characteristic of the same person.*

If we want our new environment to be a group (section 3.8.1), so that any alteration of the normal functioning of ConTEXt that happens within it disappears on leaving the environment, we must include the `\bgroup` command in the "`before`" option, and the `\egroup` command in the "`after`" option.

3.8 Other fundamental concepts

There are other notions, other than commands, that are fundamental to understanding the logic behind how ConTEXt works. Some of them, because of their complexity, are not appropriate for an introduction and therefore will not be dealt with in this document; but there are two notions that should be examined now: groups and dimensions.

3.8.1 Groups

A group is a well-defined fragment of the source file that ConTEXt uses as a *working unit* (what this means is explained shortly). Every group has a beginning and end that needs to be expressly indicated. A group begins:

- With the reserved character "`{`" or with the command `\bgroup`.
- With the command `\begingroup`
- With the command `\start`
- With the opening of certain environments (`\startSomething` command).
- By beginning a maths environment (with the reserved character «$»).

and is closed

- With the reserved character "`}`" or with the command `\egroup`.
- With the command `\endgroup`
- With the command `\stop`

- With the closing of the environment (`\stopSomething` command).
- On leaving the maths environment (with the reserved character «$»).

Certain commands also automatically generate a group, for example, `\hbox`, `\vbox` and, in general, commands linked to the creation of *boxes*[1]. Outside of these latter cases (groups automatically generated by certain commands), the way of closing a group has to be consistent with the way it is opened. This means that a group that is begun with "`{`" must close with "`}`", and a group begun with `\begingroup` must be closed with `\endgroup`. This rule has only one exception, that a group begun with "`{`" can be closed with `\egroup`, and the group begun with `\bgroup` can be closed with "`}`"; in reality, this means that "`{`" and `\bgroup` are completely synonymous and interchangeable, and similarly for "`}`" and `\egroup`.

> The commands `\bgroup` and `\egroup` were designed to be able to define commands to open a group and others to close a group. Therefore, for reasons internal to TEX syntax, those groups could not be opened and closed with curly brackets, since this would generate unbalanced curly brackets in the source file, and this would always throw an error when compiling.
>
> The commands `\begingroup` and `\endgroup`, by contrast, are not interchangeable with curly brackets or the `\bgroup ... \egroup` commands, since a group begun with `\begingroup` has to be closed with `\endgroup`. These latter commands were designed to allow for much more in-depth error checking. In general, normal users do not have to use them.

We can have nested groups (a group within another group), and in this case the order in which groups are closed must be consistent with the order in which they were opened: any subgroup has to be closed within the group in which it began. There can also be empty groups generated with "`{}`". An empty group, in principle, has no effect on the final document, but it can be useful, for example, for indicating the end of a command name.

The main effect of the groups is to encapsulate their content: as a rule, the definitions, formats and value assignments that are made within a group are "forgotten" once we leave the group. This way, if we want ConTEXt to temporarily alter its normal way of functioning, the most efficient way is to create a group and, within it, alter that functioning. Thus, when we leave the group, all the values and formats previous to it will be restored. We have already seen some examples of this when mentioning commands like `\it`, `\bf`, `\sc`, etc. But this doesn't only happen with format commands: the group in a way *isolates* its contents, so that any change in any of the many internal variables that ConTEXt is constantly managing, will remain effective only as long as we are within the group in which that change took place. Likewise, a command defined within a group will not be known outside it.

So, if we process the following example

[1] The *box* notion is also a central ConTEXt one, but its explanation is not included in this introduction.

```
\define\A{B}
\A
{
  \define\A{C}
  \A
}
\A
```

we will see that the first time we run the command \A, the result corresponds to that of its initial definition ('B'). Then we created a group and redefined the command \A within it. If we run it now within the group, the command will give us the new definition ('C' in our example), but when we leave the group in which the command \A was redefined, if we run it again it will type 'B' once more. The definition made within the group is "forgotten" once we have left it.

Another possible use of the groups concerns those commands or instructions designed to apply exclusively to the character that is written after them. In this case, if we want the command to be applied to more than one character, we must enclose the characters we want the command or instruction to be applied to, in a group. So for example, the reserved "^" character which, we already know, converts the following character into a superscript when used inside the maths environment; so if we write, for example, "`$4^2x$`" we will get "4^2x". But if we write "`$4^{2x}$`" we will get "4^{2x}".

Finally: a third use of grouping is to tell ConTEXt that what is enclosed within the group must be treated as one. This is the reason why before (section 3.5) it was said that on certain occasions it is better to enclose the contents of some command option within curly brackets.

3.8.2 Dimensions

Although we could use ConTEXt perfectly without worrying about dimensions, we would not be able to make use of all the configuration possibilities without giving them some consideration. Because to a large extent the typographical perfection achieved by TEX and its derivatives lies in the great attention that the system pays internally to dimensions. Characters have dimensions; the space between words, or between lines, or between paragraphs have dimensions; lines have dimensions; margins, headers and footers. For almost every element on the page we can think of there will be some dimension.

In ConTEXt dimensions are indicated by a decimal number followed by the unit of measurement. The units that can be used are found in table 3.2.

Name	Name in ConTEXt	Equivalent
Inch	in	1 in = 2.54 cm
Centimetre	cm	2.54 cm = 1 inch
Millimetre	mm	100 mm = 1 cm
Point	pt	72.27 pt = 1 inch
Big point	bp	72 bp = 1 inch
Scaled point	sp	65536 sp = 1 point
Pica	pc	1 pc = 12 points
Didot	dd	1157 dd = 1238 points
Cicero	cc	1 cc = 12 didots
	em	
	ex	

Table 3.2 Units of measurement in ConTEXt

The first three units in the table 3.2 are standard measures of length; the first is used in some parts of the English-speaking world and the others outside it or in some parts of it. The remaining units come from the world of typography. The last two, for which I have put no equivalent, are relative units of measurement based on the current font. 1 "em" is equal to the width of an "M" and an "ex" is equal to the height of an "x". The use of measures related to font size allows the creation of macros that look just as good whatever the source used at any given moment. That is why, in general, it is recommended.

With very few exceptions, we can use any unit of measurement we prefer, as ConTEXt will convert it internally. But whenever a dimension is indicated, it is compulsory to indicate the unit of measurement, and even if we want to indicate a measurement of "0", we have to say '0pt' or '0cm'. Between the number and the name of the unit, we may or may not leave a blank space. If the unit has a decimal part, we can use a decimal separator, either the (.) or the comma (,).

The measurements are usually used as an option for some command. But we can also directly assign a value to some internal measure of ConTEXt as long as we know the name of it. For example, indentation of the first line of an ordinary paragraph is internally controlled by ConTEXt with a variable called `\parindent`. By expressly assigning a value to this variable we will have altered the measurement that ConTEXt uses from that point on. And so, for example, if we want to eliminate the indentation of the first line, we only need to write in our source file:

```
\parindent=0pt
```

We could also have written `\parindent 0pt` (without the equal sign) or `\parindent0pt` with no space between the name of the measure and its value.

However, assigning a value directly to an internal measure is considered "inelegant". In general, it is recommended to use the commands that control that variable, and to do so in the preamble of the source file. The opposite results in source files that are very difficult to debug because not all the configuration commands are in the same place, and it is really difficult to obtain a certain consistency in typographical characteristics.

Some of the dimensions used by ConTEXt are "elastic", that is, depending on the context, they can take one or other measure. These measures are assigned with the following syntax:

```
\MeasureName plus MaxIncrement minus MaxDecrease
```

For example

```
\parskip 3pt plus 2pt minus 1pt
```

With this instruction we are telling ConTEXt to assign to `\parskip` (indicating the vertical distance between paragraphs) a *normal* measurement of 3 points, but that if the composition of the page requires it, the measurement can be up to 5 points (3 plus 2) or only 2 points (3 minus 1). In these cases it will be left to ConTEXt to choose the distance for each page between a minimum of 2 points and a maximum of 5 points

3.9 Self-learning method for ConTEXt

The huge quantity of ConTEXt commands and options turns out to be truly overwhelming and can give us the feeling that we will never end up learning to work well with it. This impression would be a mistake, because one of the advantages of ConTEXt is the uniform way it handles all its structures: learning well a few structures, and knowing, more or less, what the remaining ones are for, when we need some extra utility it will be relatively easy learn to use it. Therefore, I think of this introduction as a kind of *training* that will prepare us to make our own investigations.

To create a document with ConTEXt it is probably only essential to know the following five things (we could call them the ConTEXt *Top Five*):

1. Know how to create the source file or project of any; this is explained in Chapter 4 of this introduction.

2. Set the main font for the document, and know the basic commands to change font and colour (Chapter 6).

3. Know the basic commands for structuring the content of our document, such as chapters, sections, subsections, etc. This is all explained in Chapter 7.

4. Perhaps know how to handle the *itemize* environment explained in some detail in section 12.3.

5. ... and little else.

For the rest, all we need to know is that it is possible. Certainly no one will use a utility if they do not know that it exists. Many of them are explained in this introduction; but, above all, we can repeatedly watch how ConTEXt always acts when faced with a certain type of construction:

- First there will be a command that allows it to do so.
- Second, there is almost always a command that allows us to configure and predetermine how the task will be carried out; a command whose name starts with `\setup` and usually coincides with the basic command.
- Finally, it is often possible to create a new command to perform similar tasks, but with a different configuration.

To see whether these commands exist or not, look up the official list of commands (see section 3.6), which will also inform us of the configuration options that these commands support. And although at first glance the names of these options may seem *cryptic*, We will soon see that there are options that are repeated in many commands and that work the same or very similarly in all of them. If we have doubts about what an option does, or how it works, it will be enough to generate a document and test it. We can also look at the abundant ConTEXt documentation. As is common in the world of free software, "ConTEXt Standalone" includes the sources of almost all its documentation in the distribution. A utility like "`grep`" (for GNU Linux systems) can help us search whether the command or option that we have doubts about is used in any of these source files so that we can have an example on hand.

This is how learning ConTEXt has been conceived: the introduction explains in detail the five (actually four) aspects that I have highlighted, and many more: as we read, a clear picture of the sequence will form in our minds: *a command to carry out the task – a command that configures the previous one – a command that allows us to create a similar command.* We will also learn some of the main structures of ConTEXt, and we will know what they are for.

Chapter 4
Source files and projects

As we already know, when working with ConTEXt we always start with a text file in which, along with the contents of the final document, a number of instructions are included, telling ConTEXt about the transformations it must apply to generate our final correctly formatted document in PDF from the source file.

Thinking of the readers who until now have only known how to work with word processors, I think it is worth spending some time with the source file itself. Or rather, source files, since there are times where there is only one source file and others when we use a number of source files to arrive at the final document. In this last instance we can talk about "multifile projects".

4.1 Encoding source files

The source file(s), need to be text files. In computer terminology, this is the name given to a file containing only human-readable text which does not include binary code. These files are also called *simple text* or *plain text* files.

Since internally, computer systems only process binary numbers, a text file is really made up of *numbers* that represent *characters*. A *table* is used to connect a number with a character. For text files, there are several possible tables. The term *text file encoding* refers to the specific character-matching table that a particular text file uses.

> The existence of different encoding tables for text files is a consequence of the history of computer science itself. In the early stages of development, when the memory and storage capacity of computer devices was scarce, it was decided to use a table called ASCII (standing for "*American Standard Code for Information Interchange*") that only allowed 128 characters

and was established in 1963 by the US Standards Committee. It is obvious that 128 characters is not enough to represent all the characters and symbols used in all the languages of the world; but it was more than enough to represent English which is, of all Western languages, the one that has fewer characters, because it does not use any diacritics (accents and other marks above or below or through other letters). The advantage of using ASCII was that text files would take up very little space, as 127 (the highest number in the table) can be represented by a 7-digit binary number, and the first computers used the byte as a unit for measuring memory, an 8-digit binary number. Any character in the table would fit into a single byte. Since the byte has 8 digits and ASCII used only 7 digits, there was even space left to add some other characters to represent other languages.

But when the use of computers expanded, the inadequacy of ASCII became apparent and it became necessary to develop *alternative* tables that included characters not known to the English alphabet such as the Spanish 'ñ', accented vowels, the Catalan or French 'ç', etc. On the other hand, there was no initial agreement as to what these *alternative tables* of ASCII should be, so different specialised computer companies gradually tackled the problem on their own. Therefore, not only were specific tables created for different languages or groups of languages, but also different tables according to the company that had created them (Microsoft, Apple, IBM, etc.).

It was only with the increase in computer memory and the lower cost of storage devices and the corresponding increase in capacity that the idea of creating a single table that could be used for all languages arose. But, once again, it was not actually a single table containing all the characters that was created, but a standard encoding (called Unicode) along with different ways of representing it (UTF-8, UTF-16, UTF-32, etc.) Of all these systems, the one that has ended up becoming the de facto standard is UTF-8, which makes it possible to represent practically any living language, and many already extinct languages, as well as numerous additional symbols, all using numbers of variable length (between 1 and 4 bytes), which, in turn, helps to optimise the size of text files. This size has not increased *too much* compared to files using pure ASCII.

Up until XeTeX appeared, systems based on TeX – which was also born in the US and therefore has English as it native language – assumed the encoding was in pure ASCII; so that to use a different encoding, you had to indicate this somehow in the source file.

ConTeXt Mark IV assumes that the encoding will be UTF-8. However, in less up-to-date computer systems, a different encoding may still be used by default. I am not very sure about the default encoding that Windows uses, given that Microsoft's strategy for reaching out to the wider public consists in hiding the complexity (but even though hidden, it does not mean it has disappeared!). There is not much information available (or I have not been able to find it) regarding the encoding system it uses by default.

In any case, whatever the default encoding, any text editor allows you to save the file in the desired encoding. The source files intended to be processed by ConTeXt Mark IV must be saved in UTF-8, unless, of course, there is a very good reason for using a different encoding (although I cannot think what this reason might be).

If we want to write a file written in another encoding (perhaps an old file) we can

a. Convert the file to UTF-8, the recommended option, and there are various tools for doing this; in Linux, for example, the commands `iconv` or `recode`.

b. Tell ConTEXt in the source file that the encoding is not UTF-8. To do this we need to use the command `\enableregime`, the syntax of which is:

```
\enableregime[Encoding]
```

where *Encoding* refers to the name by which ConTEXt knows the actual encoding of the file in question. In the table 4.1 you will find the various encodings and the names that ConTEXt knows them by.

Encoding	Name(s) in ConTEXt	Notes
Windows CP 1250	cp1250, windows-1250	Western Europe
Windows CP 1251	cp1251, windows-1251	Cyrillic
Windows CP 1252	cp1252, win, windows-1252	Western Europe
Windows CP 1253	cp1253, windows-1253	Greek
Windows CP 1254	cp1254, windows-1254	Turkish
Windows CP 1257	cp1257, windows-1257	Baltic
ISO-8859-1, ISO Latin 1	iso-8859-1, latin1, il1	Western Europe
ISO-8859-2, ISO Latin 2	iso-8859-2, latin2, il2	Western Europe
ISO-8859-15, ISO Latin 9	iso-8859-15, latin9, il9	Western Europe
ISO-8859-7	iso-8859-7, grk	Greek
Mac Roman	mac	Western Europe
IBM PC DOS	ibm	Western Europe
UTF-8	utf	Unicode
VISCII	vis, viscii	Vietnamese
DOS CP 866	cp866, cp866nav	Cyrillic
KOI8	koi8-r, koi8-u, koi8-ru	Cyrillic
Mac Cyrillic	maccyr, macukr	Cyrillic
Others	cp855, cp866av, cp866mav, cp866tat, ctt, dbk, iso88595, isoir111, mik, mls, mnk, mos, ncc	Various

Table 4.1 Main encodings in ConTEXt

ConTEXt Mk IV strongly recommends the use of UTF-8. I agree with this recommendation. From here on in this introduction, we can assume that the encoding is always UTF-8.

Along with `\enableregime` ConTEXt includes the command `\useregime` which allows us to use the code for one or other encodings as an argument. I have found no information about this command nor how it differs from `\enableregime`, only some examples of its use. I suspect that `\useregime` is designed for complex projects that use many source files, with the expectation that not all of them will have the same coding. But it is only a guess.

4.2 Characters in the source file(s) that ConTEXt treats in a special way

Special characters is the name I will give to a group of characters that are different from *reserved characters*. As seen in section 3.1, they are ones that have a special meaning for ConTEXt and so cannot be used directly as characters in the source file. Along with these there is another group of characters that, although treated as such by ConTEXt when it finds them in the source file, it does treat them with special rules. This group includes blank spaces (white space), tabs, line breaks and hyphens.

4.2.1 Blank spaces (white space) and tabs

Tabs and blank spaces are treated the same in the source file for all intents and purposes. A tab character (the Tab key on the keyboard) will be transformed into white space by ConTEXt. And blank spaces are absorbed into any other blank space (or tab) immediately following them. Thus, it makes absolutely no difference in the source file to write

```
Tweedledum and Tweedledee.
```

or

```
Tweedledum   and   Tweedledee.
```

ConTEXt considers these two sentences to be exactly the same. Therefore, if we want to introduce an additional blank space between the words, we need to use some ConTEXt commands that do this. Normally it will work with "\␣", meaning a \ character followed by a blank space. But there are other procedures that will be looked at in chapter 10.3 regarding horizontal space.

The absorption of consecutive blank spaces allows us to write the source file by visually highlighting parts we would like to highlight, simply by increasing or decreasing the indentation used, with the peace of mind of knowing that it will not in any way affect the final document. Thus, in the following example

```
The music group from Madrid at the end of the seventies {\em La Romántica
  Banda Local} wrote songs of an eclectic style that were very difficult to
categorise. In their son "El Egipcio", for example, they said:
\quotation{{\em Esto es una farsa más que una comedia, página muy seria
  de la histeria musical; sueños de princesa, vicios de gitano pueden en
  su mano acariciar la verdad}}, mixing word, phrases simply because they
have an internal rhythm (comedia-histeria-seria, gitano-mano).
```

you can see how some lines are slightly indented to the right. These are lines that are part of the bits that will appear in italics. Having these indented helps (the author) to see where the italics end.

> Some might think, what a mess! Do I have to bother with indenting lines? The truth is that this special indenting is done automatically by my text editor (GNU Emacs) when it is editing a ConTEXt source file. It's that kind of small help that makes you choose to work with a certain text editor and not another one.

The rule that blank spaces are absorbed applies exclusively to consecutive blank spaces in the source file. Therefore, if an empty group ("`{}`"), is placed in the source file between two blank spaces, although the empty group will not produce anything in the final file, its presence will ensure that the two blanks are not consecutive. For example, if we write "`Tweedledum {} and Tweedledee`", we will get "Tweedledum and Tweedledee", where, if you look closely enough, you will see two consecutive spaces between the first two words.

The same happens with the reserved "`~`" character, although its effect is to generate a blank space even though it really isn't one: a blank space followed by a ~ will not absorb the latter, and a blank space after ~ will not be absorbed either.

4.2.2 Line breaks

In most text editors, when a line exceeds the maximum width, a line break is automatically inserted. We can also expressly insert a line break by pressing the "Enter" or "Return" key.

ConTEXt applies the following rules to line breaks:

a. A single line break is, to all intents and purposes, equal to a blank space. Therefore, if immediately before or after the line break there is any blank space or tab, these will be absorbed by the line break or the first blank space, and in the final document a simple blank space will be inserted.

b. Two or more consecutive line breaks create a paragraph break. For this, two line breaks are considered to be consecutive if there is nothing but blank spaces or tabs between the first and second line break (because these are absorbed by the first line break); which, in short, means that one or more consecutive lines that are absolutely blank in the source file (without any character in them, or only with blank spaces or tabs) become a paragraph break.

Note that I said "two or more consecutive line breaks" and then "one or more blank consecutive lines", meaning that if we want to increase the separation between paragraphs, we do not do so simply by inserting another line break. For this we need to use the command that increases vertical space. If we only want one extra

line of separation, we can use the \blank command. But there are other procedures for increasing vertical space. I refer to section 11.2.

On some occasions, when a line break becomes white space, we can end up with some undesirable and unexpected white space. Especially when we are writing macros, where it is easy for a blank space to "sneak in" without us realising it. To avoid this we can use the reserved character "%" which, as we know, causes the line where it appears not to be processed, which implies that the break at the end of the line will also not be processed. So, for example, the command

```
\define[3]\Test{
  {\em #1}
  {\bf #2}
  {\sc #3}
}
```

that writes its first argument in italics, the second in bold and the third in small caps, would insert a blank space between each of these arguments, while

```
\define[3]\Test{%
  {\em #1}%
  {\bf #2}%
  {\sc #3}%
}
```

will not insert any blank space between them, since the reserved character % prevents line breaks from being processed and just become a blank space.

4.2.3 Rules/dashes

Dashes are a good example of the difference between a computer keyboard and printed text. On a normal keyboard, there is usually only one character for the dash (or rule, in typographic terms) which we call the hyphen or ("-"); but a printed text uses up to four different lengths for rules:

- Short rules (hyphens), like those used to separate syllables in hyphenation at the end of a line (-).
- Medium-sized rules (en dashes or en rules), slightly longer than the previous ones (–). They have a number of uses including, for some European languages (less so in English) the beginning of a line of dialogue, or to separate the lesser from the greater digits in a range in dates or pages; "pp. 12–33".
- Long rules (em dashes or em rules) (—), used as parentheses to include one sentence within another.
- Minus sign (−) to represent subtraction or a negative number.

Today, all the above and others besides are available in UTF-8 encoding. But since they can't all be generated by a single key on the keyboard, they are not so easy to produce in a source file. Fortunately, TEX saw the need to include more

rules/dashes in our final document than could be produced by the keyboard, and designed a simple procedure to do so. ConTEXt has complemented this procedure by also adding commands that generate these various kinds of rules. We can use two approaches for generating the four kinds of rule: either the ordinary ConTEXt way with a command, or directly from the keyboard. These procedures are shown in table 4.2:

Type of rule	Appearance	Written directly	Command
Hyphen	-	`-`	`\hyphen`
En rule	–	`--`	`\endash`
Em rule	—	`---`	`\emdash`
Minus sign	−	`$-$`	`\minus`

Table 4.2 Rules/dashes in ConTEXt

The command names `\hyphen` and `\minus` are the ones normally used in English. While many in the printing industry call them 'rules', TEX's terms, namely `\endash` and `\emdash` are also common in typesetting terminology. The "*en*" and "*em*" are the names of units of measure used in typography. An "en" represents the width of an 'n' while an "em" is the width of an 'm' in the font being used.

4.3 Simple and multifile projects

In ConTEXt we can use just one source file that includes absolutely all the contents of our final document as well as all the details relating to it, in which case we are talking about "simple projects", or, by contrast, we could use a number of source files which share the contents of our final document, and in this case we are talking about "multifile projects".

The scenarios where it is typical to work with more than one source file are as follows:

- If we are writing a document in which a number of authors have collaborated, each with their own part different from the others; for example, if we are writing a festschrift with contributions from different authors, or the number of a journal, etc.
- If we are writing a lengthy document where each part (chapter) has relative autonomy, so that the final arrangement of these allows for several possibilities and will be decided at the end. This occurs with relative frequency for many academic texts (manuals, introductions and the like) where the order of chapters may vary.
- If we are writing a number of related documents that share some style characteristics.

- If, put simply, the document we are working on is large, such that the computer slows down either when editing or compiling it; in this case, splitting the material across several source files will considerably speed up the compilation for each.
- Also, if we have written a number of macros that we want to apply in some (or all) our documents, or if we have generated a template that controls or styles our documents and we want to apply these to them, etc.

4.4 Structure of the source file in simple projects

In simple projects developed in a single source file, the structure is very simple and revolves around the "`text`" environment that must essentially appear in the same file. We differentiate between the following parts of this file:

- **The document preamble**: everything from the first line in the file up to the beginning of the "`text`" environment (`\starttext`).
- **The body of the document**: this is the contents of the "`text`" environment; or in other words, everything between `\starttext` and `\stoptext`.

```
% First line of the document

% Preamble area:
% Containing the global configuration
% commands for the document

\starttext % The body of the document begins here

...
... % Document contents
...

\stoptext % End of the document
```

Figure 4.1 file containing a simple project

In figure 4.1 we see a very simple source file. Absolutely everything before the command `\starttext` (which in the image is on line 5, counting only those with some text), constitutes the preamble; everything between `\starttext` and `\stoptext` constitutes the body of the document. Anything after stoptext will be ignored.

The preamble is used to include commands that are to affect the document as a whole, the ones that determine its overall configuration. It is not essential to write

any command in the preamble. If there is none, ConTEXt will adopt a default configuration which is not very developed but could do for many documents. In a well-planned document, the preamble will contain all the commands affecting the document as a whole, like macros and customised commands to be used in the source file. In a typical preamble, this could include the following:

- An indication of the document's main language (See section 10.5).
- An indication of paper size (section 5.1) and page layout (section 5.3).
- Features of the documents main font (section 6.3).
- Customisation of the section commands to be used (section 7.4) and, if needs be, definition of new section commands (section 7.5).
- Layout of headers and footers (section 5.6).
- Settings for our own macros (section 3.7).
- Etc.

The preamble is intended for the overall configuration of the document; therefore nothing that is to do with the *contents* of the document, or processable text, should be there. In theory, any processable text included in the preamble will be ignored, although sometimes, if it is there, it will cause a compiling error.

The body of the document, framed between the `\starttext` and `\stoptext` commands includes the actual contents, meaning processable text, along with ConTEXt commands that should not affect the whole document.

4.5 Multifile management in TEX style

In order to work with more than one source file, TEX included the primitive called `\input`, which also works in ConTEXt, although the latter includes two specific commands that to some extent perfect the way `\input` functions.

4.5.1 The `\input` command

The `\input` command inserts the contents of the file it indicates. Its format is:

```
\input FileName
```

where *FileName* is the name of the file to insert. Note that it is not necessary for the file name to be enclosed between curly brackets, even though it will not throw an error if this is done. However, it should never be put between square brackets. If the file extension is "`.tex`", it can be omitted.

When ConTEXt is compiling a document and finds an `\input` command, it looks for the file indicated and continues compiling as if this file were part of the file that called it. When it finishes compiling it, it returns to the original file and continues

from where it left off; the practical result is, therefore, that the contents of the file called by means of \input are inserted at the point where that is called. The file called with \input must have a valid name in our operating system and no blank spaces within the name. ConTEXt will look for it in the working directory, and if it doesn't find it there, it will look for it in directories included in the variable of the TEXROOT environment. If the file is not ultimately found, it will produce a compilation error.

The most common use of the \input command is as follows: a file is written, let's call it "principal.tex", and this will be used as a container for calling, through the \input command, the various files that make up our project. This is shown in the following example:

```
% General configuration commands:

  \input MyConfiguration

\starttext

  \input PageTitle
  \input Preface
  \input Chap1
  \input Chap2
  \input Chap3

  ...

\stoptext
```

Note how, for the general configuration of the document, we have called the file "MyConfiguration.tex" which we assume contains the global commands we want to apply. Then, between the commands \starttext and \stoptext we call the several files that contain the contents of the various parts of our document. If, at a given moment, to speed up the compiling process, we want to leave out compiling some files, all we need to do is put a comment mark at the beginning of the line calling that or those files. For example, if we are writing the third chapter and we want to compile it simply to check that there are no errors in it, we don't need to compile the rest and therefore can write:

```
% General configuration commands:

  \input MyConfiguration

\starttext

  % \input PageTitle
  % \input Preface
  % \input Chap1
  % \input Chap2

  \input Chap3

  ...

\stoptext
```

and only Chapter 3 will be compiled. Note how, on the other hand, changing the order of chapters is as simple as changing the order of the lines calling them.

> When we exclude a file in a multifile project from being compiled, we gain in processing speed, but as a result, all the references that the part being compiled makes to other parts not as yet compiled will no longer work. See section 9.2.

It is important to be clear that when we are working with `\input`, only the main file, the one that calls all the others, must include the `\starttext` and `\stoptext` commands, because if the other files include them, there will be an error. This, on the other hand, means that we cannot directly compile the different files that make up the project, but must necessarily compile them from the main file, which is the one that houses the basic structure of the document.

4.5.2 \ReadFile and \readfile

As we have just seen, if ConTEXt does not find the file called with `\input`, it will generate an error. For the situation where we want to insert a file only if it exists, but allowing for the possibility that it might not, ConTEXt offers a variation of the `\input` command. This is

```
\ReadFile{FileName}
```

This command is similar to `\input` in every respect, with the only exception that if the file to be inserted is not found, it will continue compiling without generating any kind of error. It also differs from `\input` in its syntax, since we know that with `\input` it is not necessary to put the file name of the file to be inserted between curly brackets. But with `\ReadFile` it is necessary. If we don't use curly brackets, ConTEXt will think that the name of the file to be sought is the same as the first character that follows the `\ReadFile` command, followed by the extension `.tex`. So, for example, if we write

```
\ReadFile MyFile
```

ConTEXt will understand that the file to be read is called "M.tex", since the character immediately after the command \ReadFile (excluding blank spaces that are, as we know, ignored at the end of a command name) is an 'M'. Since ConTEXt will not normally find a file called "M.tex", and \ReadFile does not generate an error if it doesn't find the file, ConTEXt will continue compiling after the 'M' in "MyFile", and will insert the text "yFile".

A more refined version of \ReadFile is \readfile whose format is

```
\readfile{FileName}{TextIfExists}{TextIfNotExists}
```

The first argument is similar to \ReadFile: the name of a file enclosed between curly brackets. The second argument includes the text to be written if the file exists, before inserting the contents of the file. The third argument includes the text to be written if the file in question is not found. This means that depending on whether or not the file entered as the first argument is found, the second argument (if the file exists) or the third (if the file does not exist) will be executed.

4.6 ConTEXt projects as such

The third mechanism that ConTEXt offers for multifile projects is more complex and complete: it starts by distinguishing between project files, product files, component files and environment files. To understand the relations and functioning of each of these types of file, I think it is best to explain them each individually:

4.6.1 *Environment* files

An environment file is a file that stores the macros and configurations of a specific style that is intended to be applied to several documents, whether they are completely independent documents or parts of a complex document. The environment file, therefore, can include everything we would normally write before \starttext; that is: the general configuration of the document.

> I have retained the term "environment files" for these kinds of files, in order not to depart from the ConTEXt official terminology, even though I believe that a better term would probably be "format files" or "global configuration files".

Like all ConTEXt source files, the environment files are text files, and assume that the extension will be ".tex", although if we want we can change it, perhaps to ".env". Usually this is not done in ConTEXt however. Most often the environment file is identified by starting or ending the name with 'env'. For example:"MyMacros_env.tex" or "env_MyMacros.tex". The inside of such an environment file would look something like the following:

```
\startenvironment MyEnvironment

  \mainlanguage[en]

  \setupbodyfont
    [modern]

  \setupwhitespace
    [big]

  ...

\stopenvironment
```

Or in other words, definitions and configuration commands come within `\startenvironment` and `\stopenvironment`. Immediately following `\startenvironment` we write the name by which we want to identify the environment in question, and then include all the commands we would like our environment to be made up of.

> With regard to the name of the environment, according to my tests, the name we add immediately after `\startenvironment` is merely indicative, and if we were to give it no name, then nothing (bad) happens.

Environment files were intended to work with components and products (explained in the next section). This is why one or more environments can be called from a component or a product using the `\environment` command. But this command also works if it is used in the configuration area (preamble) of any ConTEXt source file, even if it is not a source file intended to be compiled in parts.

The `\environment` command can be called using either of the two following formats:

```
\environment File

\environment[File]
```

In either case, the effect of this command will be to load the contents of the file taken as an argument. If that file is not found, it will continue compiling in a normal way without generating any error. If the file extension is "`.tex`", it can be omitted.

4.6.2 Components and products

If we think of a book where each chapter is in a different source file, then we would say that the chapters are *components* and the book is the *product*. This means that the *component* is an autonomous part of a *product*, able to have its own style and to be compiled independently. Each component will have a different file, and, in addition, there will be a product file that brings all the components together.

A typical component file would be as follows

```
\environment MyEnvironment
\environment MyMacros

\startcomponent Chapter1

  \startchapter[title={Chapter 1}]

  ...

\stopcomponent
```

And a product file would look like the following:

```
\environment MyEnvironment
\environment MyMacros

\startproduct MyBook

  \component Chapter1
  \component Chapter2
  \component Chapter3

  ...

\stopproduct
```

Note that the actual contents of our document will be distributed among the various 'component' files and the product file is limited to establishing the order of the components. On the other hand, the (individual) components and the products can be compiled directly. Compiling a product will generate a PDF file containing all the components of that product. If, on the other hand, one of the components is compiled individually, it will generate a pdf file containing only the compiled component.

Within a component file, and before the `\startcomponent` command, we can call one or more environment files with `\environmentEnvironmentName`. We can do the same in the product file before `\startproduct`. Several environment files can be loaded simultaneously. We can, for example, have our favourite collection of macros and the different styles we apply to our documents all in different files. Please note, however, that when we use two or more environments, these are loaded in the order in which they are called, so that if the same configuration command has been included in more than one environment, and it has different values, the values of the last loaded environment will apply. On the other hand, environment files are loaded only once, so in the previous examples in which the environment is called from the product file and from specific component files, if we compile the product, that is the time when the environments are loaded, and in the

order indicated there; when an environment is called from any of the components, ConTeXt will check if that environment is already loaded, in which case it will do nothing.

The name of the component that is called from a product must be the name of the file that contains the component in question, although, if the extension of this file is "`.tex`", it can be omitted.

4.6.3 Projects as such

The distinction between products and components is sufficient in most cases. Just the same, ConTEXt has an even higher level where we can group a number of products: this is the *project*.

A typical project file would be more or less as follows

```
\startproject MyCollection

  \environment MyEnvironment
  \environment MyMacros

  \product Book01
  \product Book02
  \product Book03

  ...

\stopproject
```

A scenario where we would need a project would be, for example, where we need to edit a collection of books, all with the same format specifications; or if we were editing a journal: the collection of books, or the journal as such, would be the project; each book or each journal issue would be a product; and each chapter of a book or each article in a journal issue would be a component.

Projects, on the other hand, are not intended to be compiled directly. Consider that by definition each product belonging to the project (each book in the collection, or each journal issue) should be compiled separately and generate its own PDF. Therefore the `\product` command included in it to indicate what products belong to the project, actually does nothing: it is simply a reminder for the author.

Clearly, some could ask why we have projects if they can't be compiled: the answer is that the project file links certain environments to the project. This is why, if we include the `\projectProjectName` command in a component or product file, ConTEXt will read the project file and automatically load the environments linked to it. This is why the `\environment` command in projects has to come after `\startproject`; however, in products and components, `\environment` has to come *before* `\startproduct` or `\startcomponent`

Just like with the \environment and \component commands, the \project command allows us to indicate the project name either inside square brackets or not use square brackets at all. This means that \project FileName and \Project[FileName] are equivalent commands.

Summary of the different ways of loading an environment

It follows from the above that an environment can be loaded by any of the following procedures:

a. By inserting the \environment EnvironmentName command before \starttext or \startcomponent. This will load the environment for compiling the file in question only.

b. By inserting the \environment EnvironmentName command in a product file before \startproduct. This will load the environment when the product is compiled, but not if its components are compiled individually.

c. By inserting the \project command in a product or environment: this will load all environments linked to the project (in the project file).

4.6.4 Common aspects of environments, components, products and projects

Names of environments, components, products and projects: We have already seen that, for all these elements, after the \start command that initiates a particular environment, component, product or project, its name is entered directly. This name, as a rule, must coincide with the name of the file containing the environment, component or product because, for example, when ConTEXt is compiling a product and, according to the product file must load an environment or component, we have no way of knowing which file that environment or component is unless the file has the same name as the element to be loaded.

Otherwise, according to my tests, the name written after \startproduct or \startenvironment in the product and environment files is merely indicative. If it is omitted, or does not match the name of the file, nothing bad happens. However, in the case of components, it is important that the name of the component matches the name of the file that contains it.

Structure of project directories: We know that by default ConTEXt looks for files in the working directory and in the path indicated by the TEXROOT variable. However, when we use the \project, \product, \component or \environment commands it is assumed that the project has a directory structure in which common elements are found in the parent directory, and the specific

ones in some child directory. So, if the file indicated in the working directory is not found, it will be searched for in its parent directory, and if it is not found there either, in that directory's parents directory, and so on.

II
Global aspects of the document

Chapter 5
Pages and document pagination

ConTEXt transforms the source document into correctly formatted *pages*. What these pages look like, how the text and blank spaces are distributed and what elements they have, are all fundamental for good typesetting. This chapter is dedicated to all these questions, and to some other matters relating to pagination.

5.1 Page size

5.1.1 Setting page size

By default, ConTEXt assumes that documents will be of A4 size, the European standard. We can establish a different size with `\setuppapersize` that is the typical command found in the document preamble. The *normal* syntax of this command is:

```
\setuppapersize[LogicalPage][PhysicalPage]
```

where both arguments take symbolic names.[1] The first argument, that I have called *LogicalPage*, represents the page size to be taken into consideration for typesetting; and the second argument, *PhysicalPage*, represents the size of the page it will be printed on. Normally both sizes are the same, and the second argument can then be omitted; however, on occasions the two sizes can be different, as, for example, when printing a book in sheets of 8 or 16 pages (a common printing technique, especially for academic books until approximately the 1960s). In these cases, ConTEXt allows us to distinguish both sizes; and if the idea is that several pages are to be printed on a single sheet of paper, we can also indicate the folding scheme to be followed by using the `\setuparranging` command (which will not be explained in this introduction).

For typesetting size we can indicate any of the predefined sizes used by the paper industry (or by ourselves). This includes:

- Any of the A, B and C series defined by the ISO-216 (from A0 to A10, from B0 to B10 and from C0 to C10), generally in use in Europe.
- Any of the US standard sizes: letter, ledger, tabloid, legal, folio, executive.
- Any of the RA and RSA sizes defined by the ISO-217 standard.
- The G5 and E5 sizes defined by the Swiss SIS-014711 standard (for doctoral theses).
- For envelopes: any of the sizes defined by the North American standard (envelope 9 to envelope 14) or by the ISO-269 standard (C6/C5, DL, E4).
- CD, for CD covers.
- S3 – S6, S8, SM, SW for screen sizes in documents not intended to be printed but shown on screen.

Together with the paper size, with `\setuppapersize` we can indicate page orientation: "portrait" (vertical) or "landscape"(horizontal).

Other options that `\setuppapersize` allows, according to the ConTEXt wiki, are "`rotated`", "`90`", "`180`", "`270`", "`mirrored`" and "`negative`". In my own tests I have only noticed some perceptible changes with "`rotated`" that inverts the page, although it is not exactly an inversion. The numerical values are supposed to produce the equivalent degree of rotation, on their own or in combination with "`rotated`", but I have been unable to get them to work. Nor have I exactly discovered what "`mirrored`" and "`negative`" are for.

[1] Recall that in section 3.5 I indicated that the options taken by ConTEXt commands are basically of two kinds: symbolic names, whose meaning is already known to ConTEXt, or variable that we must explicitly assign some value to.

The second argument of \setuppapersize, that I have already said can be omitted when the print size is no different from the typesetting size, can take the same values as the first, indicating paper size and orientation. It can also take "oversized" as a value that – according to ConTEXt wiki – adds a centimetre and a half to each corner of the paper.

> According to the wiki there are other possible values for the second argument: "undersized", "doublesized" and "doubleoversized". But in my own tests I have not seen any change after using any of these; nor does the official definition of the command (see section 3.6) mention these options.

5.1.2 Using non-standard page sizes

If we want to use a non-standard page size, there are two things we can do:

1. Use an alternative syntax of \setuppapersize that allows us to introduce the height and width of the paper as dimensions.
2. Define our own page size, assigning a name to it and using it as if it were a standard paper size.

Alternative syntax of \setuppapersize

Other than the syntax we have already seen, \setuppapersize allows us to use this other one:

\setuppapersize[Name][Options]

where *Name* is an optional argument that represents the name assigned to a paper size with \definepapersize (that we will look at next), and *Options* are of the kind where we assign an explicit value. Among the allowable options we can highlight the following:

- **width, height** that represent, respectively, the width and height of the page.
- **page, paper**. The first refers to the size of the page to be typeset, and the second to the size of the page to be physically printed on. This means that "page" is equivalent to the first argument of \setuppapersize in its normal syntax (explained above) and "paper" to the second argument. These options can take the same values indicated earlier (A4, S3, etc.).
- **scale**, indicates a scaling factor for the page.
- **topspace, backspace, offset,** additional distances.

Defining a customised page size

To define a customised page size, we use the \definepapersize command, whose syntax is

```
\definepapersize[Name][Options]
```

where *Name* refers to the name given to the new size and *Options* can be:

- Any of the allowable values for `\setuppapersize` in its normal syntax (A4, A3, B5, CD, etc).
- Measurements of width (of the paper), height (of the paper) and offset (displacement), or a scaled value ("`scale`").

What is not possible is to mix the values allowed for `\setuppapersize` with measurements or scale factors. This is because in the first case the options are symbolic words and in the second, variables given an explicit value; and in ConTEXt, as I have already said, it is not possible to mix both kinds of options.

When we use `\definepapersize` to indicate a paper size that coincides with some of the standard measurements, in actual fact, rather than defining a new paper size, what we are doing is defining a new name for an already existing paper size. This can be useful if we want to combine a paper size with an orientation. So, for example, we can write

```
\definepapersize[vertical][A4, portrait]
\definepapersize[horizontal][A4, landscape]
```

5.1.3 Changing the page size at any point in the document

In most cases documents only have one page size and this is why `\setuppapersize` is the typical command we include in the preamble and use only once in each document. However, on some occasions it might be necessary to change the size at some point in the document; for example, if from a certain point onwards an annex is included in which the sheets are landscape.

In such cases we can use `\setuppapersize` at the precise point where we want the change to happen. But since the size would change immediately, to avoid unexpected results we would normally insert a forced page break before `\setuppapersize`.

If we only need to change the page size for an individual page, instead of using `\setuppapersize` twice, once to change to the new size, and the second to return to the original size, we can use `\adaptpapersize` that changes the page size, and, a page later, automatically resets the value prior to it being called. And just the same as we did with `\setuppapersize`, before using `\adaptpapersize` we should insert a forced page break.

5.1.4 Adjusting the page size to its contents

There are three environments in ConTEXt that generate pages of the exact size for storing their contents. These are \startMPpage, \startpagefigure and \startTEXpage. The first generates a page that contains a graphic generated with MetaPost, a graphic design language that integrates with ConTEXt, but which I will not deal with in this introduction. The second does the same with an image and perhaps some text beneath it. It takes two arguments: the first identifies the file containing the image. I will deal with this in the chapter dedicated to external images. The third (\startTEXpage) contains the text which is its contents on a page. Its syntax is:

```
\startTEXpage[Options] ... \stopTEXpage
```

where the options can be any of the following:

- **strut**. I am not sure about the usefulness of this option. In ConTEXt terminology, a *strut* is a character lacking width, but with maximum height and depth, but I don't quite see what that has to do with the overall usefulness of this command. According to the wiki this option allows for the values "yes", "no", "global" and "local", and where the default value is "no".
- **align**. Indicates text alignment. This can be "normal","flushleft", "flushright", "middle", "high", "low" or "lohi".
- **offset** to indicate the amount of white space around the text. It can be "none", "overlay" if an overlay effect is desired, or an actual dimension.
- **width, height** where we can indicate a width and height for the page, or the value "fit" so that the width and height are those required by the text that is included in the environment.
- **frame** that is "off" by default but can take the value "on" if we want a border around the text on the page.

5.2 Elements on the page

ConTEXt recognises different elements on pages, whose dimensions can be configured with \setuplayout. We will look at this immediately, but beforehand it would be best to describe each of the page elements, indicating the name that \setuplayout knows each of them by:

- **Edges:** white space surrounding the text area. Although most word processors call them "margins", using ConTEXt's terminology is preferable since it enables us to differentiate between edges as such, where there is no text (although in

electronic documents there can be navigation buttons and the like), and margins where certain text elements can sometimes be located, like, for example, margin notes.

- The height of the upper edge is controlled by two measurements: the upper edge itself ("`top`") and the distance between the upper edge and the header ("`topdistance`"). The sum of these two measurements is called "`topspace`".
- The size of the left and right edges depends on the "`leftedge`" "`rightedge`" measurements respectively. If we want both to be of the same length we can configure them simultaneously with the "`edge`" option.

  In documents intended for double-sided printing, we don't talk about left and right edges but inner and outer ones; the first is the edge closest to the point where the sheets will be stapled or sewn, i.e. the left edge on odd-numbered pages and the right edge on even-numbered pages. The outer edge is the opposite of the inner edge.
- The height of the lower edge is called "`bottom`".

- **Margins** properly so called. ConTEXt only calls side margins (left and right) margins. Margins are located between the edge and the main text area and are intended to host certain text elements such as, for example, margin notes, section titles or their numbers.

  The dimensions that control margin size are:

  - **`margin`**: used when we want to simultaneous set the margins at the same size.
  - **`leftmargin, rightmargin`**: set the size of the left and right margins respectively.
  - **`edgedistance`**: Distance between the edge and the margin.
  - **`leftedgedistance, rightedgedistance`**: Distance between the edge and the left and right margins respectively.
  - **`margindistance`**: Distance between the margin and the main text area.
  - **`leftmargindistance, rightmargindistance`**: Distance between the main text area and right and left margins respectively.
  - **`backspace`**: this measurement represents the space between the left corner of the paper and the beginning of the main text area. Therefore it is made up of the sum of "`leftedge`" + "`leftedgedistance`" + "`leftmargin`" + "`leftmargindistance`".

- **Header and footer:** The header and footer of a page are two areas that are located, respectively, in the top or bottom of the written area of the page.They usually contain information that helps to contextualise the text, such as the page number, the name of the author, the title of the work, the title of the chapter or section, etc. In ConTEXt these areas on the page are affected by the following dimensions:
  - **`header`**: Height of the header.
  - **`footer`**: Height of the footer
  - **`headerdistance`**: Distance between the header and the page's main text area.
  - **`footerdistance`**: Distance between the footer and the page's main text area.
  - **`topdistance, bottomdistance`**: Both mentioned previously. They are the distance between the upper edge and header or the lower edge and footer, respectively.
- **Main text area**: this is the widest area on the page, holding the document's text. It size depends on the "`width`" and "`textheight`" variables. The "`height`" variable, for its part, measures the sum of "`header`", "`headerdistance`", "`textheight`", "`footerdistance`" and "`footer`".

We can see all these areas in image 5.1 along with the names given to the corresponding measurements, and arrows indicating their extent. The thickness of the arrows together with the size of the names of the arrows are intended to reflect the importance of each of these distances for the page layout. If we look closely, we will see that this image shows that a page can be represented as a table with 9 rows and 9 columns, or, if we do not take into account the separation values between the different areas, there would be five rows and five columns of which there can only be text in three rows and three columns. The intersection of the middle row with the middle column constitutes the main text area and will normally take up the majority of the page.

In the layout phase of our document, we can see all the page measurements with `\showsetups`. To see the main outlines of text distribution indicated visually on the page, we can use `\showframe`; and with `\showlayout` we can get a combination of the previous two commands.

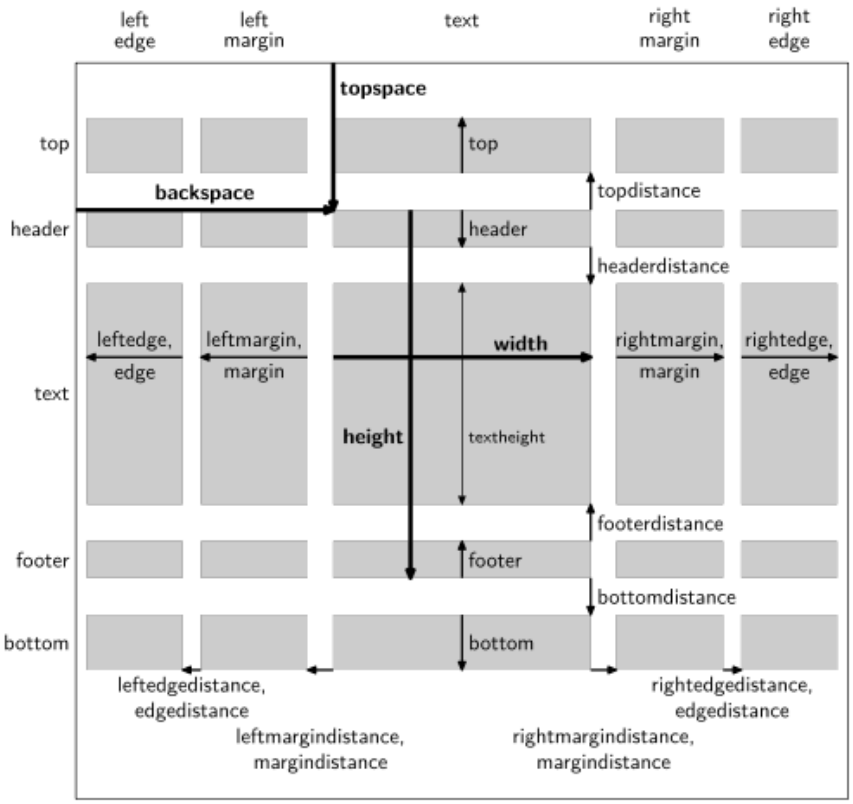


Figure 5.1 Areas and measurements on a page

5.3 Page layout (\setuplayout)

5.3.1 Assigning a size to the different page components

Page design involves assigning specific sizes to the respective areas of the page. This is done with `\setuplayout`. This command allows us to alter any of the dimensions mentioned in the previous section. Its syntax is as follows:

```
\setuplayout[Name] [Options]
```

where *Name* is an optional argument used only for the case where we have designed multiple layouts (see section 5.3.3), and the options are, besides others we will see later, any of the measurements previously mentioned. Bear in mind, however, that these measurements are inter-related since the sum total of the components affecting width, of those affecting height must coincide with the width and height of the page. In principle this will mean that when changing a horizontal length, we must adjust the remaining horizontal lengths; and the same when adjusting a vertical length.

By default, ConTEXt only carries out automatic adjustments of dimensions in some cases which, on the other hand, are not indicated in any complete or systematic way in its documentation. By carrying out several tests I was able to verify, for example, that a manual increase or decrease in the height of the header or footer entails an adjustment in "`textheight`"; however a manual change of some of the margins does not automatically adjust (according to my tests) the text width ("`width`"). This is why the most efficient way to not generate any inconsistency between the page size (set with `\setuppapersize`) and the size of its respective components, is:

- Regarding horizontal measurements:
  - By adjusting "`backspace`" (that includes "`leftedge`" and "`leftmargin`").
  - By adjusting "`width`" (text width) not with a dimensions but with the "`fit`" or "`middle`" values:
    - `fit` calculates the width of the text on the basis of the width of the rest of the page's horizontal components.
    - `middle` does the same, but first makes right and left margins equal.
- Regarding vertical measurements:
  - By adjusting "`topspace`".
  - By adjusting the "`fit`" or "`middle`" values to "`height`". These work the same way as in the case of width. The first calculates the height based on the rest of the components, and the second first makes the upper and lower margins equal, and then calculates the text height.
  - Once "`height`" is adjusted, by adjusting the height of the header or footer if necessary, knowing that in such cases "`textheight`" will be automatically readjusted.
- Another possibility for indirectly determining the height of the main text area, is by indicating the number of lines that are to fit in it (bearing in mind the current interline space and font size). This is why `\setuplayout` includes the "`lines`" option.

Placing the logical page on the physical page

In the case where the logical page size is not the same as the physical page size (see section 5.1.1) `\setuplayout` allows us to configure some additional options affecting the placement of the logical page on the physical page:

- **location**: This option determines the place where the page will be placed on the physical page. Its possible values are left, middle, right, top, bottom, singlesided, doublesided or duplex.
- **scale**: Indicates a scaling factor for the page before placing it on the physical page.
- **marking**: Will print visual marks on the page to indicate where the paper is to be cut.
- **horoffset, veroffset, clipoffset, cropoffset, trimoffset, bleedoffset, artoffset**: A series of measures indicating different displacements in the physical page. Most of these are explained in the 2013 reference manual.

These \setuplayout options must be combined with indications from \setupranging that indicates how logical pages are to be ordered on the physical sheet of paper. I will not explain these commands in this introduction, since I haven't carried out any tests on them.

Getting the width and heights of the text area

The \textwidth and \textheight commands return the width and height of the text area respectively. The values these commands offer cannot be directly shown in the final document, but they can be used for other commands to set their width or height measurements. So, for example, to indicate that we want an image whose width will be 60% of the width of the line, we need to indicate as the value of the image's "width" option: "width=0.6\textwidth".

5.3.2 Adapting the page layout

It could be that our page layout on a particular page produces an undesired result; like, for example, the final page of a chapter with only one or two lines, which is neither typographically or aesthetically desirable. To solve these cases, ConTEXt provides the \adaptlayout command that allows us to alter the size of the text area on one or more pages. This command is intended to be used only when we have already finished writing our document and are making some small final adjustments. Therefore, its natural location is in the preamble to the document. The command's syntax is:

```
\adaptlayout[Pages] [Options]
```

where *Pages* refers to the number of the page or pages whose layout we want to change. It is an optional argument that is to be used only when \adaptlayout is placed in the preamble. We can indicate just one page, or several pages, separating the numbers with commmas. If we omit this first argument, \adaptlayout will exclusively affect the page on which it finds the command.

As for the options, they can be:

- **height**: Allows us to indicate, as a dimensions, the height the page in question should have. We can indicate an absolute height (e.g. "19cm") or a relative height (e.g. "+1cm", "-0.7cm").
- **lines**: We can include the number of lines to add or subtract. To add lines the value is preceded by a +, and to subtract lines, by the − sign (not just a hyphen).

Consider that when we change the number of lines on a page, this could affect the pagination of the rest of the document. this is why it is recommended that we use `\adaptlayout` only at the end, when the document will not have further changes, and to do it in the preamble. Then we go to the first page we wish to adapt, do so and check how it affects the pages that follow; if it affects it in such a way that another page needs adapting, we add its number and compile once again, and so on.

5.3.3 Using multiple page layouts

If we need to use different layouts in different parts of the document, the best way is to begin by defining the *general* layout and then the various alternative ones, those that only change the dimensions that need to be different. These alternative layouts will inherit all the features of the overall layout which will not change as part of its definition. To specify an alternative layout and give it a name we can later call it with, we use the `\definelayout` command whose general syntax is:

```
\definelayout[Name/Number] [Configuration]
```

where *Name/Number* is the name associated with the new design, or the page number where the new layout will be automatically activated, and *Configuration* will contain the aspects of the layout that we wish to change by comparison with the overall layout.

When the new layout is associated with a name, to call it at a particular point in the document we use:

```
\setuplayout[LayoutName]
```

and to return to the general layout:

```
\setuplayout[reset]
```

If, on the other hand, the new layout was associated with a specific page number, it will be automatically activated when the page is reached. However, once activated, to return to the general design we will have to explicitly indicate this, even though

we can *semi-automate* this. For example, if we want to apply a layout exclusively to pages 1 and 2, we can write in the document's preamble:

```
\definelayout[1][...]
\definelayout[3][reset]
```

The effect of these commands will be that the layout defined in the first line is activated on page 1 and on page 3 another layout is activated the function of which is only to return to the general layout.

With `\definelayout[even]` we create a layout that is activated on all even pages; and with `\definelayout[odd]` the layout will be applied to all odd pages.

5.3.4 Other matters related to page layout

A. Distinguishing between odd and even pages

In double-sided printed documents it is often the case that the header, page numbering and side margins differ between odd and even pages. Even-numbered pages are also called left hand (verso) pages and odd pages, right hand (recto) pages. In these cases it is also usual for the terminology regarding margins to change, and we talk about inner and outer margins. The former is located at the closest point to where the pages will be sewn or stapled and the latter on the opposite side. On odd-numbered pages, the inner margin corresponds to the left margin and on even pages the outer margin corresponds to the right margin.

`\setuplayout` does not have any option expressly allowing us to tell it that we want to differentiate between the layout for odd and even pages. This is because for ConTEXt the difference between both kinds of pages is set with a different option: `\setuppagenumbering` that we will see in section 5.4. Once this has been set, ConTEXt assumes that the page described with `\setuplayout` was the odd page, and builds the even page by applying the inverted values for the odd page to it: the specifications applicable on the odd-numbered page apply to the left, on the even-numbered page they apply to the right; and vice versa: those applicable on the odd-numbered page on the right, apply to the even-numbered page on the left.

B. Pages with more than one column

With `\setuplayout` we can also see that the text of our document is distributed across two or more columns, in the way that newspapers and some magazines do, for example. This is controlled by the "`columns`" option the value of which has to be a whole number. When there is more than one column, the distance between the columns is indicated by the "`columndistance`" option.

This option is intended for documents in which all the text (or most of it) is distributed across multiple columns. If, in a document that is mainly a one column document, we want a particular part to be two or three columns, we do not need to alter the page layout but simply use the "`columns`" environment (see section 12.2).

5.4 Page numbering

By default, ConTEXt uses Arabic numbers for page numbering and the number appears centred in the header. To alter these features, ConTEXt it has different procedures which, in my opinion, make it unnecessarily complex where this matter is concerned.

Firstly, the fundamental characteristics of numbering are controlled by two different commands: `\setuppagenumbering` and `\setupuserpagenumber`.

`\setuppagenumbering` allows the following options:

- `alternative`: This option controls whether the document is designed so that the header and footer are identical on all pages ("`singlesided`"), or whether they differentiate odd and even pages ("`doublesided`"). When this option takes the latter value, automatically the page layout values introduced by "`setuplayout`" are affected, so that it is assumed that what is indicated in "`setuplayout`" refers only to odd-numbered pages, and therefore what is arranged for the left margin actually refers to the inner margin (which on even-numbered pages is on the right) and that what is arranged for the right side actually refers to the outer margin, which on even-numbered pages is on the left.

- `state`: Indicates whether or not the page number will be displayed. It allows two values: start (page number will be displayed) and stop (page numbers will be suppressed). The name of these values (start and stop) could make us think that when we have "`state=stop`" pages stop being numbered, and when "`state=start`" numbering begins again. But this is not so: these values only affect whether the page number is shown or not.

- `location`: indicates where it will be displayed. Normally we need to indicate two values in this option, separated by a comma. First of all we need to specify if we want the page number in the header ("`header`") or the footer ("`footer`"), and then, where in the header or footer: it could be "`left`", "`middle`", "`right`", "`inleft`", "`inright`", "`margin`", "`inmargin`", "`atmargin`" or "`marginedge`". For example: to show right-aligned numbering in the footer we should indicate "`location={footer,right}`". See, on the other hand, how we have surrounded this option with curly brackets so ConTEXt can correctly interpret the separating comma.

- **style**: indicates font size and style to be used for page numbers.
- **color**: Indicates the colour to be applied to the page number.
- **left**: picks up the command or text to be executed to the left of the page number.
- **right**: picks up the command or text to be executed to the right of the page number.
- **command**: picks up a command to which the page number will be passed as a parameter.
- **width**: indicates the width taken up by the page number.
- **strut**: I am not so sure about this. In my tests, when "strut=no", the number is printed exactly on the upper edge of the header or on the bottom of the footer, while when "strut=yes" (default value) a space is applied between the number and the edge.

\setupuserpagenumber, allows these extra options:

- **numberconversion**: controls the kind of numbering that can be arabic ("n", "numbers"), lower case ("a", "characters"), upper case ("A","Characters"), small caps ("KA"), lower case roman ("i", "r", "romannumerals"), uppercase roman ("I", "R", "Romannumerals") or small caps roman ("KR").
- **number**: indicates the number to assign to the first page, on the basis of which the rest will be calculated.
- **numberorder**: if we assign "reverse" to this as a value, page numbering will be in decreasing order; this means the last page will be 1, the second-last 2, etc.
- **way**: allows us to indicate how numbering will proceed. It can be: byblock, bychapter, bysection, bysubsection, etc.
- **prefix**: allows us to indicate a prefix to page numbers.
- **numberconversionset**: Explained in what follows.

In addition to these two commands, it is also necessary to take into account the control of numbers involving the document's macrostructure (see section 7.6). From this point of view, \defineconversionset allows us to indicate a different kind of numbering for each of the macrostructure blocks. For example:

```
\defineconversionset
  [frontpart:pagenumber][][romannumerals]

\defineconversionset
  [bodypart:pagenumber][][numbers]

\defineconversionset
  [appendixpart:pagenumber][][Characters]
```

will see that the first block in our document (frontmatter) is numbered with lower case Roman numbers, the central block (bodymatter) with Arabic numbers and the appendices with upper case letters.

We can use the following commands to get the page number:

- `\userpagenumber`: returns the page number just as it was configured with `\setuppagenumbering` and with `\setupuserpagenumber`.
- `\pagenumber`: returns the same number as the previous command but still in Arabic numbers.
- `\realpagenumber`: returns the real number of the page in Arabic numbers without taking any of these specifications into account.

To get the number of the final page in the document there are three commands that are parallel to the previous ones. They are: `\lastuserpagenumber`,`\lastpagenumber` and `\lastrealpagenumber`.

5.5 Forced or suggested page breaks

5.5.1 The \page command

The algorithm for text distribution in ConTEXt is quite complex, and is based on a multitude of calculations and internal variables that tell the program where the best possible point is for introducing an actual page break from the perspective of typographical correctness. The `\page` command allows us to influence this algorithm:

a. By suggesting certain points as the best or the most inappropriate place for including a page break.

- **no**: indicates that the place where the command is located is not a good candidate for inserting a page break, so, as far as possible, the break should be made at another point in the document.
- **preference**: tells ConTEXt that the point where it encounters the command is a *good place* for attempting a page break, although it will not force one there.
- **bigpreference**: indicates that the point where it encounters the command is a *very good place* for attempting a page break, but it too does not go as far as forcing it.

Note that these three options neither force nor prevent page breaks, but only tell ConTEXt that when looking for the best place for a page break, it should take into account what is indicated in this command. In the final instance, however, the place where the page break will happen will continue to be decided by ConTEXt.

b. By forcing a page break at a certain point; in this case we can also indicate how many page breaks should be made as well as certain features of the pages to be inserted.

- **yes**: force a page break at this point.
- **makeup**: similar to "yes", but the forced break is immediate, without first placing any floating objects whose placement is pending (see section 13.1).
- **empty**: insert a completely blank page in the document.
- **even**: insert as many pages as necessary to make the next page an even page.
- **odd**: insert as many pages as necessary to make the next page an odd page.
- **left, right**: similar to the two previous options, but applicable only to double-sided printed documents, with different headers, footers or margins depending on whether the page is odd or even.
- **quadruple**: insert the number of pages needed for the next page to be a multiple of 4.

Along with these options which specifically control pagination, \page includes other options that affect the way this command functions. Especially the "disable" option that causes ConTEXt to ignore the \page commands it finds from there on, and the "reset" option that produces the opposite effect, restoring the effectiveness of future \page commands.

5.5.2 Joining certain lines or paragraphs to prevent a page break from being inserted between them

Sometimes, if we want to prevent a page break between several paragraphs, the use of the \page command can be laborious, as it would have to be written at every

point where it was possible for a page break to be inserted. A simpler procedure for this is to place the material we want to keep on the same page in what TEX calls a *vertical box*.

> At the beginning of this document (on page 20) I indicated that internally, everything is a *box* for TEX. The box notion is fundamental in TEX for any kind of *advanced* operation; but managing it is too complex to include in this introduction. This is why I only make occasional references to boxes.

TEX boxes, once created, are indivisible, meaning that we cannot insert a page break that would split a box in two. This is why, if we put the material we want kept together in an invisible box, we avoid a page break being inserted that would split this material. The command for doing this is `\vbox`, the syntax for which is

```
\vbox{Material}
```

where *Material* is the text we want to keep together.

Some of ConTEXt's environments do put their contents in a box. For example, "`framedtext`", so if we frame the material we want kept together in this environment and also see that the frame is invisible (which we do with the `frame=off` option), we will have achieved the same thing.

5.6 Headers and footers

5.6.1 Commands for establishing the content of headers and footers

If we have assigned a certain size to the header and footer in page layout, we can include text in them with the `\setupheadertexts` and `\setupfootertexts` commands. The two commands are similar, the only difference being that the former activates header content and the latter the footer content. Both have from one to five arguments.

1. Used with a single argument this will contain the text of the header or footer that will be placed in the centre of the page. For example: `\setupfooter-texts[pagenumber]` will write the page number at the centre of the footer.

2. Used with two arguments, the content of the first argument will be placed on the left side of the header or footer, and that of the second argument on the right side. For example `\setupheadertexts[Preface][pagenumber]` will typeset a page header in which the word "preface" is written on the left side and the page number is printed on the right side.

3. If we use three arguments, the first will indicate *the area* in which the other two are to be printed. By *area* I am referring to the *areas* of the page mentioned in

section 5.2, in other words: edge, margin, header... The other two arguments contain the text to be placed in the left edge, margin and right edge, margin.

Using it with four or five arguments is equivalent to using it with two or three arguments, in cases where a distinction is made between even and odd pages, which occurs, as we know, when "`alternative=doublesided`" with `\setuppagenumbering` has been set. In this case, two possible arguments are added to reflect the content of the left and right sides of the even pages.

An important characteristic of these two commands is that when they are used with two arguments, the previous central header or footer (if it existed) is not rewritten, which allows us to write a different text in each area as long as we first write the central text (calling the command with a single argument) and then write the texts for either side (calling it again, now with two arguments). So, for example, if we write the following commands

```
\setupheadertexts[and]
\setupheadertexts[Tweedledum][Tweedledee]
```

The first command will write "and" in the centre of the header and the second will write "Tweedledum" on the left and "Tweedledee" on the right, leaving the centre area untouched, since it has not been ordered to be rewritten. The resulting header will now show up as

Tweedledum and Tweedledee

The explanation I have just given of the operation of these commands is my conclusion after many tests. The explanation of these commands provided in ConTEXt *excursion* is based on the version with five arguments; and the one in the 2013 reference manual is based on the version with three arguments. I think mine is clearer. On the other hand, I have not seen an explanation of why the second command call does not overwrite the previous call, but this is how it works if we first write the central item in the header or footer and then the ones either side. But if we write the items either side first in the header/footer, the subsequent call to the command to write the central item will delete the previous headers or footers. Why? I have no idea. I think these small details introduce unnecessary complication and should be clearly explained in the official documentation.

Moreover, we can indicate any combination of text and commands as the actual content of the header or footer. But also the following values:

- **`date, currentdate`**: will write (either of them) the current date.
- **`pagenumber`**: will write the page number.
- **`part, chapter, section...`**: will write the title corresponding to part, chapter, section... or whatever structural division there is.
- **`partnumber, chapternumber, sectionnumber...`**: will write the number of the part, chapter, section... or whatever structural division there is.

Attention: These symbolic names (`date, currentdate, pagenumber, chapter, chapternumber`, etc.) are only interpreted as such if the symbolic name itself is the only content of the argument; but if we add some other text or formatting command, these words will be interpreted literally, and so, for example, if we write `\setupheadertexts[chapternumber]` we will get the number of the current chapter; but if we write `\setupheadertexts[Chapter chapternumber]` we will end up with: "Chapter chapternumber". In these cases, when the content of the command is not just the symbolic word, we must:

- For `date, currentdate` and `pagenumber` use, not the symbolic word but the command with the same name (`\date`, `\currentdate` or `\pagenumber`).
- For `part, partnumber, chapter, chapternumber`, etc. use the `\getmarking[Mark]` command that returns the contents of the *Mark* that is asked for. So, for example, `\getmarking[chapter]` will return the title of the current chapter, while `\getmarking[chapternumber]` will return the number of the current chapter.

To disable headers and footers on a particular page, use the `\noheaderandfooterlines` command that acts exclusively on the page where it is located. If we only want to delete the page number on a particular page, we must use the `\page[blank]` command.

5.6.2 Formatting headers and footers

The specific format in which the text of the header or footer is shown can be indicated in the arguments for `\setupheadertexts` or `\setupfootertexts` by using the corresponding format commands. However, we can also configure this globally with `\setupheader` and `\setupfooter` that allow the following options:

- **`state`**: allows for the following values: `start, stop, empty, high, none, normal` or `nomarking`.
- **`style, leftstyle, rightstyle`**: configuration of the header and footer text style. `style` affects all pages, `leftstyle` the even pages and `rightstyle` the odd pages.
- **`color, leftcolor, rightcolor`**: header or footer colour. It can affect all pages (`color` option) or only the even pages (`leftcolor`) or odd pages (`rightcolor`)
- **`width, leftwidth, rightwidth`**: width of all headers and footers (`width`) or headers/footers on even pages (`leftwidth`) or odd ones (`rightwidth`).
- **`before`**: command to be executed before writing the header or footer.

- **after**: command to be executed after writing the header or footer.
- **strut**: if "yes", a vertical separation space is established between the header and the edge. When it is "no", the header or footer runs up against the edges of the upper or lower edge areas.

5.6.3 Defining specific headers and footers and linking them to section commands

ConTEXt's header and footer system allows us to automatically change the text in the header or footer when we change chapters or sections; or when we change pages, if we have set different headers or footers for odd and even pages. But what it does not allow is to differentiate between the first page (of the document, or of a chapter or section) and the rest of the pages. To achieve the latter we must:

1. Define a specific header or footer.
2. Link it to the section it applies to.

The definition of specific headers or footers is done with the `\definetext` command, whose syntax is:

```
\definetext
  [Name] [Type]
  [Content1] [Content2] [Content3]
  [Content4] [Content5]
```

where *Name* is the name assigned to the header or footer we are dealing with; *Type* can be `header` or `footer`, depending on which of the two we are defining, and the remaining five arguments contain the contents we want for the new header or footer, in a similar way to how we have seen `\setupheadertexts` and `\setupfootertexts` function. Once we have done this, we need to link the new header or footer to some particular section with `\setuphead` by using the *header* and `footer` options (that are not explained in Chapter 7).

Thus, the following example will hide the header on the first page of each chapter and a centred page number will appear as the footer:

```
\definetext[ChapterFirstPage] [footer] [pagenumber]
\setuphead
  [chapter]
  [header=high, footer=ChapterFirstPage]
```

5.7 Inserting text elements in page edges and margins

The top and bottom edges and the right and left margins usually do not contain text of any kind. However, ConTEXt allows some text elements to be placed there. In particular, the following commands are available for this purpose:

- `\setuptoptexts`: allows us to place text at the top edge of the page (above the header area).
- `\setupbottomtexts`: allows us to place text at the bottom edge of the page (below the footer area).
- `\margintext`, `\atleftmargin`, `\atrightmargin`, `\ininner`, `\ininneredge`, `\ininnermargin`, `\inleft`, `\inleftedge`, `\inleftmargin`, `\inmargin`, `\inother`, `\inouter`, `\inouteredge`, `\inoutermargin`, `\inright`, `\inrightedge`, `\inrightmargin`: allow us to place text in the side edges and margins of the document.

The first two commands function exactly like `\setupheadertexts` and `\setupfootertexts`, and the format of these texts can even be configured in advance with `\setuptop` and `\setupbottom` similar to how `\setupheader` allows us to configure the texts for `\setupheadertexts`. For all this I refer to what I have already said in section 5.6. The only little detail that needs to be added is that the text set up for `\setuptoptexts` or `\setupbottomtexts` will not be visible if no space has been reserved in the page layout for the upper (`top`) or lower (`bottom`) edges. For this, see section 5.3.1.

As for the commands aimed at placing text in the margins of the document, they all have a similar syntax:

```
\CommandName[Reference][Configuration]{Text}
```

where *Reference* and *Configuration* are optional arguments; the first is used for possible cross-referencing and the second allows us to set up the marginal text. The last argument, enclosed in curly brackets, contains the text to be placed in the margin.

Of these commands, the more general one is `\margintext` as it allows text to be placed in any of the margins or side edges of the page. The remaining commands, as their name indicates, place the text in the margin itself (right or left, inner or outer), or the edge (right or left, inner or outer). These commands are closely related to page layout because if, for example, we use `\inrightedge` but have not reserved any space in the page layout for the right edge, nothing will be seen.

The configuration options for \margintext are as follows:

- **location**: indicates what margin the text will be placed in. It can be `left`, `right` or, in double-sided documents, `outer` or `inner`. By default it is `left` in single-sided documents and `outer` in double-sided ones.
- **width**: width available for printing the text. By default, the full width of the margin will be used.
- **margin**: indicates whether the text will be placed in the `margin` itself or in the `edge`.
- **align**: text alignment. The same values are used here as in \setupalign 11.6.1.
- **line**: allows us to indicate a number of lines of displacement of the text in the margin. So, `line=1` will displace the text by one line below and `line=-1` by one line above.
- **style**: command or commands for indicating the style of text to be placed in the margins.
- **color**: the colour of marginal text.
- **command**: name of a command to which the text to be placed in the margin will be passed as an argument. This command will be executed before writing the text. For example, if we want to draw a frame around the text, we could use "`[command=\framed]{Text}`".

The remaining commands allow the same options, except for `location` and `margin`. In particular, the \atrightmargin and \atleftmargin commands place the text completely attached to the body of the page. We can establish a separation space with the `distance` option, which I did not mention when talking about \margintext because I saw no effect on that command in my tests.

In addition to the above options, these commands also support other options (`strut, anchor, method, category, scope, option, hoffset, voffset, dy, bottomspace, threshold and stack`) that I have not mentioned because they are not documented and frankly, I am not very sure what they are for. Ones with names like *distance* we can guess, but the rest? The wiki only mentions the `stack` option, saying that it is used to emulate the \marginpars command in LaTeX, but this does not seem very clear to me.

The \setupmargindata command allows us to globally configure the texts in each margin. So, for example,

```
\setupmargindata[right][style=slanted]
```

will ensure that all texts in the right margin are written in slanted style.

We can also create our own customised command with

```
\definemargindata[Name][Configuration]
```

Chapter 6
Fonts and colours in ConTEXt

6.1 Typographical fonts included in "ConTEXt Standalone"

ConTEXt's fonts system offers many possibilities, but it also quite complex. I will not be analysing all the advanced font possibilities in this manual, but will limit myself to assuming we are working with some of the 21 fonts provided with the installation of ConTEXt Standalone, the ones shown in table 6.1.

The central column of table 6.1 indicates the name or names by which ConTEXt knows the font in question. When there are two names, they are synonymous. The last column has an example of the font in use. As for the order in which the fonts are shown, the first is the font that ConTEXt uses by default, and the remaining fonts are in alphabetical order, while the last three fonts are specifically designed for mathematics. Note that the Euler font cannot directly represent accented letters, so we get Bront's, not Brontë's.

For readers coming from the Windows world and its default fonts, I will indicate that *heros* is equivalent to Arial in Windows, while *termes* is the same as Times New Roman. They are not exactly the same but similar enough, to the point where one would need to be very observant to tell the difference.

Official name	Name(s) in ConTEXt	Example
Latin Modern	modern, modern-base	Emily Brontë's book
Antykwa Poltawskiego	antykwapoltawskiego	Emily Brontë's book
Antykwa Toruńska	antykwa	Emily Brontë's book
Cambria	cambria	Emily Brontë's book
DejaVu	dejavu	Emily Brontë's book
DejaVu Condensed	dejavu-condensed	Emily Brontë's book
Gentium	gentium	Emily Brontë's book
Iwona	iwona	Emily Brontë's book
Latin Modern Variable	modernvariable, modern-variable	Emily Brontë's book
PostScript	postscript	Emily Brontë's book
TeX Gyre Adventor	adventor, avantgarde	Emily Brontë's book
TeX Gyre Bonum	bonum, bookman	Emily Brontë's book
TeX Gyre Cursor	cursor, courier	Emily Brontë's book
TeX Gyre Heros	heros, helvetica	Emily Brontë's book
TeX Gyre Schola	schola, schoolbook	Emily Brontë's book
Tex Gyre Chorus	chorus, chancery	*Emily Brontë's book*
Tex Gyre Pagella	pagella, palatino	Emily Brontë's book
Tex Gyre Termes	termes, times	Emily Brontë's book
Euler	eulernova	Emily Bront's book
Stix2	stix	Emily Brontë's book
Xits	xits	Emily Brontë's book

Table 6.1 Fonts included in the ConTEXt distribution

Fonts used by Windows are not *free software* (in fact almost nothing in Windows is *free software*), so they cannot be included in a ConTEXt distribution. However, if ConTEXt is installed in Windows, then these fonts are already installed and can be used like any other font installed on the system running ConTEXt. In this introduction, though, I will not deal with how to use fonts already installed on the system. Help can be found for this on the ConTEXt wiki.

6.2 Font features

6.2.1 Fonts, *styles* and style variants

The terminology regarding fonts is somewhat confusing, since at times what is called a font is really a *font family* that includes different styles and variants that share a basic design. I will not enter into the question of which terminology is the more correct; I am only interested in clarifying the terminology used in ConTEXt. There, it makes a distinction between fonts, styles and variants (or alternatives) for each style. The *fonts* included in the ConTEXt distribution (in fact they are *font families*) are the ones we saw in the previous section. We will look now at *styles* and *alternatives*.

Font styles

Donald E. Knuth designed the *Computer Modern* font for TEX, giving it three distinct *styles* called *roman*, *sans serif* and *teletype*. The *roman* style is a design

where the characters have decorative flourishes known in typological terminology as *serifs*, which is why this font style is also known as *serif*. This style was considered to be the *normal* or default style. The *sans serif* style, as its name indicates, lacks these flourishes, and hence is a simpler, more stylised font, sometimes known by other names, e.g. in Spanish, *paloseco*; this font can be the principal font in the document, but it is also appropriate for use in certain fragments of a text whose principal font is in *roman* style, like, for example, title or page headers. Finally, the *teletype* style was included in *Computer Roman* since this had been designed for writing books to do with computer programming, involving large sections in computer *code* which is conventionally represented, in printed material, in a monospaced style that imitates computer terminals and the old typewriters.

A fourth style intended for maths fragments could be added to these three font *styles*. But since TEX automatically uses this style when it enters maths mode, and does not include commands to expressly enable or disable it, nor does it have the *variants* or alternatives of the other styles, it is not usual to think of it as a *style* properly so called.

ConTEXt includes commands for two possible additional styles: handwritten and calligraphic. I am not exactly sure about the difference between them since, on the one hand, none of the fonts included in the ConTEXt distribution include these styles in their design, and on the other hand, as I see it, calligraphic writing is also handwritten. These commands that ConTEXt includes to enable such styles, if used with a font that does not implement them, will not cause any error when compiling: it is simply that nothing happens.

Alternative font forms

Each *style* allows a number of alternative forms, and that is what ConTEXt calls them, (*alternative*):

- Regular or normal ("`tf`", from *typeface*).
- Bold ("`bf`", from *boldface*).
- Italic ("`it`" from *italic*)
- BoldItalic ("`bi`" from *bold italic*)
- Slanted ("`sl`" from *slanted*)
- BoldSlanted ("`bs`" from *bold slanted*)
- Small caps ("`sc`" from *small caps*)
- Medieval ("`os`" from *old style*)

These *alternatives*, as their name indicates, are mutually exclusive: when one is enabled, the others are disabled. This is why ConTEXt provides commands for enabling them but not for disabling them; because when we enable an alternative, we disable the one we were using until then; and so, for example, if we are writing in italic and enable bold, then italic will be disabled. If we want to use bold and italic simultaneously, we do not have to enable one and then the other, but rather enable the alternative that includes both ("`bi`").

On the other hand, it must be born in mind that although ConTEXt assumes that every font will have these alternatives, and therefore provides commands to enable them, in order to function and produce some perceptible effect in the final document, these commands need the font to have specific forms in their design for each style and alternative.

In particular, many fonts do not differentiate in their design between slanted and italic letters, or do not include special forms for small caps.

Difference between italics and slanted

The similarity in the typographical function performed by italics and slanted letters leads many people to confuse these two alternatives. The slanted letter is obtained by slightly rotating the regular shape. But italics implies – at least in certain fonts – a different design in which the letters *seem* to be tilted because they have been drawn to look like it; but in reality there is no authentic tilt. This can be seen in the following example, in which we have written the same word three times at the same size large enough to make it easy to appreciate the differences. In the first version the regular form is used, in the second the slanted, and in the third italics:

italics – *italics* – *italics*

Note how the design of the characters is the same in the first two examples, but in the third there are subtle differences in the strokes of some letters, which is very obvious, especially in how the 'a' is drawn, although the differences actually occur in almost all characters.

The usual uses of italic and slanted letters are similar and each person decides whether to use one or the other. Here there is freedom, although we should point out that a document will be better typeset and will look better if the use of italic and slanted lettering is *consistent*. In many fonts, moreover, the design difference between italic and slanted is negligible, so it makes no difference whether we use one or the other.

On the other hand, both italic and slanted are font alternatives, which mainly means two things:

1. We can only use them when they are defined in the font.
2. When enabling one of them we are disabling the alternative that was being used up until then.

Together with the commands for italic and slanted, ConTEXt offers an additional commands for *emphasising* a particular text. Its use implies subtle differences by comparison with italic or slanted. See section 6.5.1.

6.2.2 Font size

All the fonts handled by ConTEXt are based on vector graphics, so that in theory they can be displayed at any font size, although as we will see, this depends on the actual instructions we use to determine font size. Unless otherwise stated, it is assumed that the font size will be 12 points.

> All fonts used by ConTEXt are based on vector graphics, and are therefore Opentype or Type 1 fonts, which implies that fonts whose origins predate this technology have been reimplemented. In particular, the TEX default font, *Computer Modern*, designed by Knuth, only existed in certain sizes, so was reimplemented in a design called *Latin Modern* used by ConTEXt, although in many documents it continues to be called *Computer Modern* due to the strong symbolism that font still has for TEX systems, since these started out and were developed by Knuth along with another program called MetaFont, aimed at designing fonts that could work with TEX.

6.3 Setting the document's main font

By default, unless some other font is indicated, ConTEXt will use *Latin Modern Roman* at 12 point as the main font. This font was originally designed by KNUTH to be implemented in TEX. It is an elegant roman-style font with great proportional and decorative "flourishes" – called *serifs* – in certain strokes, which is very appropriate both for printed texts and for display on screen; although – and this is a personal opinion – it is not so suitable for small screens like the *smartphone*, because the *serifs* or flourishes tend to pile up, making reading difficult.

To set up a different font we use `\setupbodyfont` that allows us not only to change the actual font, but also its size and style. When we want this to apply to the whole document, we need to include it in the source file's preamble. But if we simply wish to change the font at a certain point, this is where we need to include what follows.

The `\setupbodyfont` format is:

```
\setupbodyfont[Options]
```

where the command's various options allow us to indicate:

- **The font name**, that can be any of the symbolic font names found in table 6.1.
- **The size**, which can be indicated either by its dimensions (using the point as the unit of measurement) or by certain symbolic names. But note that even though earlier I said that fonts used by ConTEXt can be scaled to practically any size, in `\setupbodyfont` only sizes consisting of whole numbers between 4 and 12, as well as the values 14.4 and 17.3, are supported in ConTEXt. By default it assumes the size is 12 points.

\setupbodyfont, establishes what we could call the *base size* of the document; in other words the *normal* character size on the basis of which other sizes are calculated, for example titles and footnotes. When we change the main size with \setupbodyfont all other sizes calculated on the basis of the main font are also changed.

Besides directly indicated the character size (10pt, 11pt, 12pt, etc.) we can also use some symbolic names that calculate the character size to apply, based on the current size. The symbolic names in question are, from largest to smallest: big, small, script, x, scriptscript and xx. So, for example, if we want to set body text with \setupbodyfont which is larger than 12 points, we can do so with "big".

- **font style**, which, just as we have indicated, can be roman (with serifs), or without serifs (san serif), or typewriter style, and for some fonts, handwritten and calligraphic style. \setupbodyfont allows different symbolic names to indicate different styles. These are found in table 6.2:

Style	Symbolic names allowed
Roman	rm, roman, serif, regular
Sans Serif	ss, sans, support, sansserif
Monospaced	tt, modo, type, teletype
Handwritten	hw, handwritten
Calligraphic	cg, calligraphic

Table 6.2 Styles in setupbodyfont

As far as I can tell, the different names supported for each of the styles are completely synonymous.

See what a font looks like

Before deciding to use a particular font in our document, we would normally want to see what it looks like. This can almost always be done from the operating system as there is usually some utility to examine the appearance of the fonts installed on the system; but for convenience, ConTEXt itself offers a utility that allows us to see the appearance of any of the fonts enabled in ConTEXt. This is \showbodyfont, that generates a table with examples of the font we indicate.

The format of \showbodyfont is as follows:

```
\showbodyfont[Options]
```

where we can indicate as options precisely the same symbolic names as in \setupbodyfont. So, for example, \showbodyfont[schola, 8pt] will show us the table below, in which there are different examples of the schola font at a base size of 8 points:

[schola] [schola,8pt]													
	\tf	\tf	\bf	\sl	\it	\bs	\bi	\tfx	\tfxx	\tfa	\tfb	\tfc	\tfd
\rm	Ag	Ag	**Ag**	*Ag*	*Ag*	***Ag***	***Ag***	Ag	Ag	Ag	Ag	Ag	Ag
\ss	Ag	Ag	**Ag**	*Ag*	*Ag*	***Ag***	***Ag***	Ag	Ag	Ag	Ag	Ag	Ag
\tt	Ag	Ag	**Ag**	*Ag*	*Ag*	***Ag***	***Ag***	Ag	Ag	Ag	Ag	Ag	Ag
\mr	Ag	Ag	**Ag**	*Ag*	*Ag*	***Ag***	***Ag***	Ag	Ag	Ag	Ag	Ag	Ag

Note that there are certain commands in the first row and column of the table. Further on, when the meaning of these commands has been explained, we will take another look at the tables generated by \showbodyfont.

If we want to see the complete range of characters in a specific font, we can use the \showfont[FontName] command. This command will show the main design of each of the characters without applying commands for styles and alternatives.

6.4 Changing font or some font features

6.4.1 The \setupbodyfont and \switchtobodyfont commands

To change font, style or size, we can use the same command with which we established the font at the beginning of the document, when we don't want to use ConTEXt's default font: \setupbodyfont. All we need to do is to place this command at the point in the document where we wish to change font. It will produce a *permanent* change of font, meaning that it will directly affect the main font, and indirectly all fonts related to it.

Very similar to \setupbodyfont is \switchtobodyfont. Both commands allow us to change the same aspects of the font (the font itself, style and size) but internally they function differently and are intended for different uses. The first one, (\setupbodyfont) is meant for *establishing* the main font (and normally the only one) in the document; it is neither common nor typographically correct for a document to have more than one main font (which is why it is called *main* font). By contrast, \switchtobodyfont is intended for writing some parts of a text in a different font, or to assign a particular font to a special kind of paragraph we want to define in our document.

Apart from the above – which actually affects the internal functioning of each of these two commands – from the user's point of view there are some differences between the use of one or the other command. In particular:

1. As we already know \setupbodyfont is limited to a particular range of sizes, whereas \switchtobodyfont allows us to indicate practically any size, so that if the font is not available in that size, it will scale to it.

2. \switchtobodyfont does not affect text elements in any way other than where it is used, unlike \setupbodyfont which, as mentioned above, establishes the main font and, by altering this, also alters the font of all textual elements whose font is calculated on the basis of the main font.

Both commands, on the other hand, change not only the font, style and size, but also other aspects associated with the font such as, for example, interline space.

> \setupbodyfont generates a compiling error if a non-allowed font size is requested; but does not generate one if a non-existent font is requested, in which case the default (*Latin Modern Roman*) font will be enabled. \switchtobodyfont acts the same way with regard to the font, and in terms of size, as I have already said, tries to achieve this by scaling the font. However, there are fonts that cannot be scaled to certain sizes, in which case the default font would once again be enabled.

6.4.2 Quickly changing style, alternative and size

Changing style and alternative

As well as \switchtobodyfont, ConTEXt provides a set of commands that allow us to quickly change the style, alternative or size. With regard to these commands, the ConTEXt wiki warns us that sometimes, when they appear at the beginning of a paragraph, they can produce some unwanted side effects, so it recommends that in such cases the command in question be preceded by the \dontleavehmode command.

Style	Commands that enable it
Roman	\rm, \roman, \serif, \regular
Sans Serif	\ss, \sans, \support, \sansserif
Monospaced	\tt, \mono, \teletype,
Handwritten	\hw, \handwritten,
Calligraphic	\cf, \calligraphic

Table 6.3 Commands for changing between different styles

Table 6.3 contains the commands that allow us to change style, without altering any other aspect; and table 6.4 contains the commands that allow us to exclusively alter the alternative.

All these commands retain their effectiveness until another style or alternative is explicitly enabled, or the *group* within which the command is declared ends. Therefore, when we want the command to affect only a part of the text, what we must do is to enclose that part within a group, as in the following example, where

Alternative	Commands that enable it
Normal	\tf, \normal
Italic	\it, \italic
Bold	\bf, \bold
Bold-italic	\bi, \bolditalic, \italicbold
Slanted	\sl, \slanted
Bold-slanted	\bs, \boldslanted, \slantedbold
Small caps	\sc, \smallcaps
Medieval	\os, \mediaeval

Table 6.4 Commands for enabling a particular alternative

each time the word *thought* appears when it is a noun, not a verb, it is in italics, creating a group for it.

```
I thought a {\it thought} but
the {\it thought} I thought wasn't
the {\it thought} I thought I thought.
If the {\it thought} I thought I thought
had been the {\it thought} I thought
I wouldn't have thought so much!
```

I thought a *thought*, but the *thought* I thought, wasn't the *thought* I thought I thought. If the *thought* I thought I thought had been the *thought* I thought I wouldn't have thought so much!

Suffixes for changing alternative and size at the same time

The commands that change style or alternative in their two-letter version (\tf, \it, \bf, etc.) allow a range of *suffixes* that affect font size. The suffixes a, b, c and d increase the font size, multiplying it by 1.2, 1.2^2 (= 1.44), 1.2^3 (= 1.728) or 1.2^4 (= 2.42) respectively. See an example:

```
\tf test, \tfa test, \tfb test, \tfc test, \tfd test
```

test, test, test, test, test

the suffixes x and xx reduce font size, multiplying it by 0.8 and 0.6 respectively:

```
\tf test, \tfx test, \tfxx test
```

test, test, test

The suffixes ‘x’ and ‘xx’ applied to \tf allow us to shorten the command, so that \tfx can be written as \tx and \tfxx as \txx.

The availability of these different suffixes depends on the actual implementation of the font. According to the ConTEXt 2013 reference manual (intended mostly for Mark II) the only suffix guaranteed to always work is ‘x’, and the others might or might not be implemented; or they might be just for some alternatives.

At any rate, to avoid doubts, we can use \showbodyfont that I spoke of previously (in section). This command displays a chart that not only allows us to appreciate the appearance of the font, but also to see what the font looks like in each of its styles and alternatives, as well as what resizing suffixes are available.

Let us look at the table showing \showbodyfont once more:

[modern]													
	\tf	\tf	\bf	\sl	\it	\bs	\bi	\tfx	\tfxx	\tfa	\tfb	\tfc	\tfd
\rm	Ag	Ag	**Ag**	*Ag*	*Ag*	***Ag***	***Ag***	Ag	Ag	Ag	Ag	Ag	Ag
\ss	Ag	Ag	**Ag**	*Ag*	*Ag*	***Ag***	***Ag***	Ag	Ag	Ag	Ag	Ag	Ag
\tt	Ag	Ag	**Ag**	*Ag*	*Ag*	***Ag***	***Ag***	Ag	Ag	Ag	Ag	Ag	Ag
\mr	Ag	Ag	**Ag**	*Ag*	*Ag*	***Ag***	***Ag***	*Ag*	*Ag*	*Ag*	*Ag*	*Ag*	*Ag*

If we look closely at the table, we can see that the first column contains the font styles (\rm, \ss and \tt). The first row contains, on the left, the alternatives (\tf, \sc, \sl, \it, \bf, \bs and \bi), while the right side of the first row contains the other available suffixes, although only with the regular alternative.

It is important to note that a change in font size made by any of these suffixes will only change the font size in the strict sense, leaving intact other values normally associated with font size such as line spacing.

Customising the scaling factor of the suffixes

To customise the scaling factor we can use \definebodyfontenvironment whose format can be:

```
\definebodyfontenvironment[particular size][scaled]
\definebodyfontenvironment[default][scaled]
```

In the first version we would redefine the scaling for a particular size of the main font set by \setupbodyfont or by \switchtobodyfont. For example:

```
\definebodyfontenvironment[10pt][a=12pt,b=14pt,c=2, d=3]
```

would ensure that when the main font is 10 points, suffix ‘a’ would change it to 12 points, suffix ‘b’ to 14, suffix ‘c’ would multiply the original font by 2 and suffix ‘d’ by 3. Note that for a and b a fixed dimension has been indicated, but for c and d a multiplication factor of the original size has been indicated.

But when the first argument of \definebodyfontenvironment is equal to "default", then we will be redefining the scaling value for all possible font sizes, and as a scaling value we can only enter a multiplier number. So if, for example, we write:

```
\definebodyfontenvironment[default][a=1.3,b=1.6,c=2.5,d=4]
```

we will be indicating that whatever the size of the main font, the a suffix should be multiplied by 1.3, the b by 1.6, the c by 2 and the d by 4.

As well as the suffixes xx, x, a, b, c and d, with \definebodyfontenvironment we can assign a scaling value to the "big", "small", "script" and "scriptscript" key words. These values are assigned to all sizes associated with these key words in \setupbodyfont and \switchtobodyfont. They are also applied in the following commands, whose usefulness can be deduced (I think) from their name:

- \smallbold
- \smallslanted
- \smallboldslanted
- \smallslantedbold
- \smallbolditalic
- \smallitalicbold
- \smallbodyfont
- \bigbodyfont

If we want to see the default sizes of a particular font, we can use \showbodyfontenvironment[Font]. This command, applied to the modern font, for example, gives the following result:

[modern]						
text	script	scriptscript	x	xx	small	big
10pt	7pt	5pt	8pt	6pt	8pt	12pt
11pt	8pt	6pt	9pt	7pt	9pt	12pt
12pt	9pt	7pt	10pt	8pt	10pt	14.4pt
14.4pt	11pt	9pt	12pt	10pt	12pt	17.3pt
17.3pt	12pt	10pt	14.4pt	12pt	14.4pt	20.7pt
20.7pt	14.4pt	12pt	17.3pt	14.4pt	17.3pt	20.7pt
4pt	4pt	4pt	4pt	4pt	4pt	6pt
5pt	5pt	5pt	5pt	5pt	5pt	7pt
6pt	5pt	5pt	5pt	5pt	5pt	8pt
7pt	6pt	5pt	6pt	5pt	5pt	9pt
8pt	6pt	5pt	6pt	5pt	6pt	10pt
9pt	7pt	5pt	7pt	5pt	7pt	11pt

6.4.3 Defining commands and key words for font sizes, styles and alternatives

The predefined commands for changing font size, styles and variants are enough. Furthermore, ConTEXt allows us:

1. To add our own command for changing font style, size or variant.

2. To add synonyms to style or variant names recognised by `\switchtobodyfont`.

It provides the following commands to do this:

- `\definebodyfontswitch`: allows us to define a command to change font size. For example, if we want to define the `\eight` command (or the `\viiicommand`[1]) to set an 8 pt font we need to write:

  `\definebodyfontswitch[eight][8pt]` or `\definebodyfontswitch[viii][8pt]`

- `\definefontstyle`: allows us to define one or more words that can be used in `\setupbodyfont` or `\switchtobodyfont` to set a particular font style; so, for example, if we wanted to call the *sans serif* something else (e.g. in Spanish it is called “paloseco”) we can write

[1] Remember that except for the case of control symbols, ConTEXt command names can only consist of letters.

```
\definefontstyle[paloseco][ss]
```

A peculiarity of \definefontstyle is that it allows several words to be associated simultaneously with the same style, so, to continue the Spanish example:

```
\definefontstyle[paloseco, sosa, sinrebordes][ss]
```

- \definealternativestyle: allows us to associate a name with a font variant. This name could function as a command or be recognised by the `style` option of the commands that allow us to configure the style to be applied. So, for example, the following fragment

  ```
  \definealternativestyle[strong][\bf][]
  ```

  will enable the \strong command and the key word "`strong`" that will be recognised by the `style` option of the commands that allow this option. We could have said "bold" but this word is already in use for ConTeXt, so I have chosen a term used in HTML, namely, "strong" as an alternative

I do not know what the third argument of \definealternativestyle does. It is not optional and therefore cannot be omitted; but the only information I found on it is in the ConTEXt reference manual where this third argument is said to be relevant only to chapter and section titles "*where, apart from \cap, we must obey the font used here*" (??)

6.5 Other matters relating to the use of some alternatives

Among the different alternatives of a font, there are two whose use requires certain clarifications:

6.5.1 Italic, slanted and emphasis

Both italics and slanted letters are used mainly for typographically highlighting a fragment of the text to draw attention to it. In other words, to *emphasise it*.

We can, of course, emphasise a text by explicitly enabling italic or slanted. But ConTEXt offers an alternative command that is much more useful and interesting and is intended specifically for emphasising a text fragment. This is the \em command from the word *emphasis*. By contrast to \it and \sl, that are purely typographical commands, \em is a *conceptual* command; it works differently, so is more versatile, to the point where the ConTEXt documentation recommends using \em in preference to \it or \sl. When we use these two latter commands we are telling ConTEXt what font alternative we want to use; but when we use \em we are telling it what effect we want produced, leaving it up to ConTEXt to decide how to do this. Normally, to achieve the effect of emphasising or highlighting something

we would enable italic or slanted, but this depends on the context. So if we use \em in a text that is already in italic – or is slanted – the command will highlight that in the opposite way – in upright text in this case.

Hence the following example:

```
{\em One of the most beautiful
orchids in the world is the
{\em Thelymitra variegata}
or Southern Queen of Sheba.}
```

One of the most beautiful orchids in the world is the Thelymitra variegata *or Southern Queen of Sheba.*

Note that the first \em enables italics (actually, slanted, but see below) and that the second \em disables this and instead puts the words "Thelymitra variegata" in normal upright style.

Another advantage of \em is that it is not an alternative, so does not disable the alternative we had before and so, for example, in a text that is in bold, with \em we will get bold slanted without the need to explicitly call on \bs. Similarly, if the \bf command appears in a text that is emphasised already, this emphasis will not cease.

By default \em enables slanted rather than italic, but we can change this with \setupbodyfontenvironment[default][em=italic].

6.5.2 Small caps and fake small caps

Small caps is a typographical resource that is often much better than using upper case (capital) letters. Small caps give us the shape of the capital letter but keep the height the same as lower case letters on the line. This is why small caps is a stylistic variant of lower case. Small caps replace capital letters in certain contexts, and are especially useful for writing Roman numerals, or chapter titles. In academic texts it is also customary to use small caps to write the names of the authors cited.

The problem is that not all fonts implement small caps, and those that do, do not always do so for some of their *font styles*. Moreover, as small caps are an alternative to italic, bold or slanted, in accordance with the general rules we have set out in this chapter, all these typographical features could not be used simultaneously.

These problems can be resolved by using *fake small caps* that ConTEXt allows us to create with the \cap and \Cap commands; in this regard see section 10.2.1.

6.6 Use and configuration of colours

ConTEXt provides commands for changing the colour of an entire document, some of its elements, or certain parts of the text. It also provides commands for uploading

hundreds of predefined colours into memory, and for seeing what their components are.

6.6.1 Procedures for typesetting text fragments in colour

Most of ConTEXt's configurable commands allow an option called "`color`" that allows us to indicate the colour in which the text affected by that command should be written. Thus, for example, to indicate that chapter titles are written in blue, we only need to write:

```
\setuphead
  [chapter]
  [color=blue]
```

Using this procedure we can colour titles, headings, footnotes, margin notes, bars and lines, tables, table or image titles, etc. The advantage of using this procedure is that the final result will be consistent (all texts that fulfil the same function will be written with the same colour) and easier to change globally.

We can also colour a portion or fragment of text directly, although, to avoid a too-variegated use of colours, which is not pleasant from a typographical perspective, or an inconsistent use, in general it is recommended to avoid direct colouring and to use what we could call *semantic colouring*, that is, instead of, for example, writing

```
\color[red]{Very important text}
```

we define a command for very important text that is given a colour. For example

```
\definehighlight[important][color=red]
\important{Very important text}
```

6.6.2 Changing the document's background and foreground colour

If we want to change the colour of the whole document, depending on whether we want to alter the colour of the background or the colour of the foreground (text), we will use `\setupbackgrounds` or `\setupcolors`. So, for example

```
\setupbackgrounds
  [page]
  [background=color,backgroundcolor=blue]
```

This command will set the background colour of pages as blue. As a value for "`backgroundcolor`" we can use the name of any of the predefined colours.

To globally change the foreground colour throughout the document (from the point where the command is inserted) use `\setupcolors`, where the "`textcolor`" option controls the text colour. For example:

```
\setupcolors[textcolor=red]
```

will see that the text colour is red.

6.6.3 Commands for colouring particular text fragments

The general command for colouring small portions of text is

```
\color[ColourName]{Text to colour}
```

For larger portions of text it is preferable to use

```
\startcolor[ColourName] ... \stopcolor
```

Both are named after some predefined colour. If we want to define the colour on the fly, we can use the `\colored` command. For example:

```
Three \colored[r=0.1, g=0.8, b=0.8]
{coloured} cats
```

Three coloured cats.

6.6.4 Predefined colours

ConTEXt loads the most common predefined colours listed in table 6.5.[1]

Name	Light tone	Medium tone	Dark tone
black			
white			
gray	lightgray	middlegray	darkgray
red	lightred	middlered	darkred
green	lightgreen	middlegreen	darkgreen
blue	lightblue	middleblue	darkblue
cyan		middlecyan	darkcyan
magenta		middlemagenta	darmagenta
yellow		middleyellow	darkyellow

Table 6.5 ConTEXt's predefined colours

[1] This list can be found in the reference manual and ConTEXt wiki but I am fairly sure it is an incomplete list since in this document, for example, without having loaded any additional colour, we use "orange" – which is not in the table 6.5– for section titles.

There are other colour collections not loaded by default but which can be loaded with the command

```
\usecolors[CollectionName]
```

where CollectionName can be

- "`crayola`", 235 colours imitating marker shades.
- "`dem`", 91 colours.
- "`ema`", 540 colour definitions based on colours used by Emacs.
- "`rainbow`", 91 colours for use in maths formulas.
- "`ral`", 213 colour definitions from the *Deutsches Institut für Gütesicherung und Kennzeichnung* (German Institute for Quality Assurance and Labelling).
- "`rgb`", 223 colours.
- "`solarized`", 16 colours based on the solarized scheme.
- "`svg`", 147 colours.
- "`x11`", 450 standard colours for X11.
- "`xwi`", 124 colours.

> The colour definition files are included in the "`context/base/mkiv`" directory of the distribution and its name responds to the "`colo-imp-NOMBRE.mkiv`" scheme. The information I have just provided on the different collections of predefined colours is based on my particular distribution. The specific collections, or the number of colours defined in them, could change in future versions.

To see what colours each of these collections contains we can use the `\showcolor[CollectionName]` command described in what follows. To use some of these colours we first need to load them into memory with the (`\usecolors[CollectionName]`) command and then we have to tell the `\color` or `\startcolor` commands the name of the colour. For example the following sequence:

```
\usecolors[xwi]
\color[darkgoldenrod]{Tweedledum and Tweedledee}
```

will write

Tweedledum and Tweedledee

6.6.5 To see available colours

The `\showcolor` command shows a list of colours in which you can see the appearance of the colour, its appearance when the colour is used in grey scale, the red, green and blue components of the colour, and the name by which ConTEXt knows it. Used without any argument `\showcolor` will show the colours used in the current document. But as an argument we can indicate any of the predefined

collections of colours that were discussed in section 6.6.4, and so, for example, `\showcolor[solarized]` will show us the 16 solarized colours in that collection:

0.561	0.514	0.580	0.588	base0
0.460	0.396	0.482	0.514	base00
0.409	0.345	0.431	0.459	base01
0.162	0.027	0.212	0.259	base02
0.123	0.000	0.169	0.212	base03
0.615	0.576	0.631	0.631	base1
0.909	0.933	0.910	0.835	base2
0.965	0.992	0.965	0.890	base3
0.457	0.149	0.545	0.824	blue
0.487	0.165	0.631	0.596	cyan
0.510	0.522	0.600	0.000	green
0.429	0.827	0.212	0.510	magenta
0.422	0.796	0.294	0.086	orange
0.395	0.863	0.196	0.184	red
0.473	0.424	0.443	0.769	violet
0.530	0.710	0.537	0.000	yellow

If we want to see the rgb components of a particular colour, we can use `\showcolorcomponents[ColourName]`. This is useful if we are trying to define a specific colour, to see the composition of some colour that is close to it. For example, `\showcolorcomponents[darkgoldenrod]` will show us:

```
color          name           transparency  specification
white black    darkgoldenrod                r=0.720,g=0.530,b=0.040
```

6.6.6 Defining our own colours

`\definecolor` allows us to either clone an existing colour or define a new colour. Cloning an existing colour is as simple as creating an alternative name for it. To do this you would have to write:

```
\definecolor[New colour][Old colour]
```

This will ensure that "*New colour*" is exactly the same colour as "*Old colour*".

But the main use of `\definecolor` is for creating new colours. To do so the command must be used in the following way:

```
\definecolor[ColourName][Definition]
```

where *Definition* can be done by applying up to six different colour generation schemes:

- **RGB colours**: The definition of RGB colours is one of the most widespread; it is based on the idea that it is possible to represent a colour by mixing, by addition, the three primary colours: red ('r' for *red*), green ('g' for *green*) and blue ('b' for *blue*). Each of these components is indicated as a decimal number between 0 and 1.

  `\definecolor[lime 1][r=0.75, g=1, b=0]`: Text in "lime 1".

- **Hex colours**: This way of representing the colours is also based on the RGB scheme, but the red, green and blue components are indicated as a three-byte hexadecimal number in which the first byte represents the value of red, the second the value of green and the third the value of blue. For example:

  `\definecolor[lime 2][x=BFFF00]`: Text in "lime 2".

- **CMYK colours**: This model of colour generation is what is called a "subtractive model" and is based on the mixture of pigments of the following colours: cyan ('c'), magenta ('m'), yellow ('y', from *yellow*) and black ('k', from *key*). Each of these components is indicated as a decimal number between 0 and 1:

  `\definecolor[lime 3][c=0.25, m=0, y=1, k=0]`: Text in "lime 3".

- **HSL/HSV**: This colour model is based on measuring the hue ('h', from *hue*), saturation ('s') and luminescence ('l' or sometimes 'v', from *value*). Hue corresponds to a number between 0 and 360; saturation and luminescence must be a decimal number between 0 and 1. For example

  `\definecolor[lime 4][h=75, s=1, v=1]`: Text in "lime 4"

- **HWB colours**: The HWB model is a suggested standard for CSS4 which measures the hue ('h', from *hue*), and the level of white ('w', from *whiteness*) and black ('b', from *blackness*). Hue corresponds to a number between 0 and 360, while whiteness and blackness are represented by a decimal number between 0 and 1.

  `\definecolor[Azure][h=75, w=0.2, b=0.7]` Text in "Azure".

- **Greyscale**: based on a component called ('s', from *scale*) that measures the amount of grey. It needs to be a number between 0 and 1. For example:

  `\definecolor[light grey][s=0.65]`: Text in "light grey".

It is also possible to define a new colour from another colour. For example, the colour in which titles are written in this introduction is defined as

```
\definecolor[maincolour][0.6(orange)]
```

Chapter 7
Document structure

7.1 Structural divisions in documents

Except for very short texts (like a letter, for example), a document is usually structured into blocks or text groupings that generally follow a hierarchical order. There is no standard way of naming these blocks: in novels, for example, the structural divisions are usually called "chapters" although some – the longer ones – have larger blocks usually called "parts" that group a number of chapters together. Theatrical works distinguish between "acts" and "scenes". Academic manuals are divided (sometimes) into "parts" and "lessons", "topics" or "chapters" which in turn often have internal divisions as well; the same kind of complex hierarchical divisions often exist in other academic or technical documents (such as texts like the present one dedicated to explaining a computer program or system. Even laws are structured into "books" (the longest and most complex, such as Codes), "titles","chapters", "sections", "subsections". Scientific and technical documents can also reach up to six, seven or on occasions even eight levels of nesting depth for these kinds of divisions.

This chapter focuses on analysing the mechanism ConTEXt offers for supporting these structural divisions. I will refer to them with the overall term of "sections".

> There is no clear term that allows us to refer generically to all these kinds of structural divisions. The term "section", that I have opted for, focuses on structural division rather than anything else, though a drawback is that one of ConTEXt's predetermined structural divisions is called a "section". I hope it does not cause confusion, believing that it will be easy enough to determine from the context if we are speaking of section as a generic and overall reference to structural divisions, or of a specific division which ConTEXt calls a section.

Each “section” (generically speaking) implies:

- A reasonably large *structural division of a document* which may, in turn, include other lower-level divisions. From this perspective “sections” imply text blocks with a hierarchical relationship between them. From the point of view of its sections, the document as a whole can be viewed as a tree. The document *per se* is the trunk, each of its chapters a branch, which in turn can have twigs that can also subdivide and so on.

  Having a clear structure is very important for the document to be read and understood. This task is up to the author, however, not the typesetter. And although it is not up to ConTEXt to make us better authors than we are, the full range of section commands it includes, where the hierarchy among them is quite clear, could help us to write better-structured documents.

- A *structure name* that we could call its “title” or “label”. This structure name is printed:

  – Always (or almost always) at the point in the document where the structural division begins.

  – At times also in the table of contents, in the header or footer of the pages occupied by the section in question.

  ConTEXt allows us to automate all these tasks in such a way that the formatting features with which the title of a structural unit should be printed only have to be indicated once, and whether it should or should not also be included in the table of contents, or in the headers or footers. To do this ConTEXt only needs to know where each structural unit begins and ends, what it is called and what hierarchical level it is at.

7.2 Section types and their hierarchy

ConTEXt distinguishes between *numbered* and *unnumbered* sections. The former, as their name suggests, are numbered automatically and sent to the table of contents, as well as, sometimes, to page headers and/or footers.

ConTEXt has hierarchically-ordered predefined section commands found in table 7.1.

With regard to the predefined sections, the following clarifications should be made:

Level	Numbered sections	Unnumbered sections
1	\part	–
2	\chapter	\title
3	\section	\subject
4	\subsection	\subsubject
5	\subsubsection	\subsubsubject
6	\subsubsubsection	\subsubsubsubject
...	...	...

Table 7.1 Section commands in ConTEXt

- In table 7.1 the section commands are shown in their traditional form. But we will immediately see that they can also be used as *environments* (`\startchapter ... \stopchapter`, for example) and that this is the approach that is actually recommended.

- The table contains only the first 6 section levels. In my tests, however, I found up to 12 levels: After `\subsubsubsection` comes `\subsubsubsubsection`, and so on as far as `\subsubsubsubsubsubsubsubsubsubsection`, or `\subsubsubsubsubsubsubsubsubsubsubject`.

  But we should bear in mind that the kind of (excessively deep) lower levels indicated above are hardly likely to improve comprehension of a text! First of all we are likely to have large sections inevitably dealing with several matters and this will make it difficult for the reader to *grasp* their content. Going to excessive depth in levels can also mean that the reader loses an overall sense of the text, and the effect produced is one of excessive fragmentation of the material involved. My understanding is that in general, four levels are sufficient; very occasionally one might need to go to six or seven levels, but any greater depth would rarely be a good idea.

  From the perspective of writing the source file, the fact that to create further sub-levels means adding yet another "sub" to the previous level can make the source file almost unreadable: it is no joke trying to work out the level of a command named "subsubsubsubsubsection" since I have to count all the "subs"! So my advice is that if we really need so many levels of depth, from the fifth level onwards (subsubsection) we would be better off defining our own section commands (see section 7.5) giving them names that are clearer than the predefined ones.

- The highest section level (`\part`) only exists for numbered titles and has the peculiarity that the part title is not printed. However, even if the title is not printed, a blank page is introduced (on which we can assume that the title is printed once the user has reconfigured the command) and the numbering of the *part* is taken into account to calculate the numbering of the chapters and other sections.

  The reasons why the default version of `\part` does not print anything is because, according to the ConTEXt wiki, almost always the title at this level requires a specific layout; and while this is true, it doesn't seem a good enough reason to me, since, in practice, chapters

and sections are also often redefined, and the fact that the parts do not print anything forces the novice user to *dip* into the documentation to see what is going wrong.

- Although the first sectioning level is the "part", this is only theoretical and abstract. In a specific document, the first sectioning level will be the one corresponding to the first sectioning command in the document. That is, in a document that does not include parts but chapters, chapter will be the first level. But if the document does not include chapters either, only sections, the hierarchy for that document will start with the sections.

7.3 Syntax common to section commands

All section commands, including any levels created by the user (see section 7.5), allow the following alternative forms of syntax (if, for example, we are using the "`section`" level):

```
\section [Label] {Title}
\section [Options]
\startsection [Options] [Variables] ... \stopsection
```

In the three ways above, arguments between square brackets are optional and can be omitted. We will look at them separately, but first of all it helps to make it clear that in Mark IV it is the third of these three approaches that is recommended.

- In the first syntax form, which we could call the "*classic*" one, the command takes two arguments, one optional between square brackets, and the other obligatory between curly brackets. The optional argument is there to associate the command with a label that will be used for internal references (see section 9.2). The obligatory one between curly brackets is the section title.

- The other two forms of syntax are more the ConTEXt style: everything the command needs to know is communicated through values and options introduced between square brackets.

  Recall that in sections 3.3.1 and 3.4 I said that in ConTEXt, the scope of the command is indicated in curly brackets, and its options in square brackets. But if we think about it, the title of a particular sectioning command is not the scope of its application, so to be consistent with the general syntax, it should not be introduced between curly brackets, but as an option. ConTEXt allows for this exception because it is the classic way of doing things in TEX, but it provides the alternative forms of syntax that are more consistent with its overall design.

  The options are of the value assignment kind (OptionName=Value), and are as follows:

- **reference**: Label for cross-references.
- **title**: The section title that will be printed in the body of the document.
- **list**: The section title that will be printed in the table of contents.
- **marking**: The section title to be printed in page headers or footers.
- **bookmark**: The section title to be converted into a *bookmark* in the PDF file.
- **ownnumber**: This option is used in the case of a section that is not automatically numbered; in this case, this option will include the number assigned to the section in question.

Of course, the "**list**", "**marking**" and "**bookmark**" options should only be used if we want to use a different title to replace the main title set with the "**title**" option. This is very useful, for example, when the title is too long for the header; although to achieve this we can also use the `\nomarking` and `\nolist` commands (something very similar). On the other hand, we need to bear in mind that if the title text (the "title" option) includes any commas, then it will need to be enclosed within curly brackets, both the complete text and the comma, to ensure that ConTeXt knows that the comma is part of the title. The same applies to the options: "list", "marking" and "bookmark". Therefore, in order not to have to keep an eye on whether or not there are any commas in the title, I think it is a good idea to get into the habit of always enclosing the value of any of these options between curly brackets.

So, for example, the following lines will create a chapter entitled "A Test Chapter" associated with the "test" label for cross-references, while the header will be "Chapter test" instead of "A Test Chapter".

```
\chapter
  [
    title={A Test Chapter},
    reference={test},
    marking={Chapter test}
  ]
```

The `\startSectionType` syntax turns the section into an *environment*. It is more consistent with the fact that, as I said at the beginning, in the background each section is a differentiated block of text, although ConTEXt, by default, does not consider *environments* generated by section commands to be *groups*. Just the same, this procedure is what Mark IV recommends; quite possibly because this way of establishing sections requires us to expressly state where each section begins and ends, which makes it easier for the structure to be consistent, and most probably

has better support for XML and EPUB output. In fact, for XML output, it is essential.

When we use `\startSectionName` one or more variables are allowed as arguments between square brackets. Their value can then be used later at other points in the document with the `\structureuservariable` command.

> Having user variables allows for very advanced uses in ConTEXt by dint of the fact that decisions can be taken regarding whether or not to compile a fragment, or in what we way to do so, or with what template depending on the value of a particular variable. These ConTEXt utilities, however, go beyond the scope of the material I wish to deal with in this introduction.

7.4 Format and configuration of sections and their titles

7.4.1 The `\setuphead` and `\setupheads` commands

By default, ConTEXt assigns certain features to each level of sectioning that mainly (but not only) affect the format in which the title is displayed in the main body of the document, but not the way the title is displayed in the table of contents or headers and footers. We can change these features with the `\setuphead` command, whose syntax is:

```
\setuphead[Sections][Options]
```

where

- **`Sections`** refers to the name of one or more sections (separated by commas) to be affected by the command. This can be:
  - Any of the predefined sections (part, chapter, title, etc.), in which case we can refer to them either by name or by their level. To refer to them by their level we use the word "`section-`*`NumLevel`*", where *NumLevel* is the level number of the section concerned. So, "`section-1`" is equal to "`part`", "`section-2`" is equal to "`chapter`", etc.
  - Any kind of section we ourselves have defined. In this regard, see section 7.5.
- **`Options`** are the configuration options. These are of the explicit value assignment kind (OptionName=value). The number of eligible options is very high (over sixty) and I will therefore explain them by grouping them into categories according to their function. I must point out, however, that I have not managed to determine what some of these options are for or how they are used. I will not talk about those options.

Previously I said that \setuphead affects the sections that are expressly indicated. But this does not mean that the modification of a particular section should not affect the others in any way unless they have been expressly mentioned in the command. In fact, the opposite is true: the modification of a section affects other sections *that are linked* to it, even if this has not been made explicit in the command. The linkage between the different sections is of two kinds:

- The unnumbered commands are linked to the corresponding numbered command of the same level so that a change in the appearance of the numbered command will affect the unnumbered command of the same level; but not the other way around: the change in the unnumbered command does not affect the numbered command. This means, for example, that if we change some aspect of "chapter" (level 2) we are also changing that aspect in "title"; but changing "title" will not affect "chapter".

- The commands are linked hierarchically, such that if we change *certain features* in a particular level, the change will affect all levels that come after it. This only happens with certain features. Colour, for example: if we establish that subsections will display in red, we are also changing subsubsections, subsubsubsections, etc. to red. But the same does not happen with other features like font style for example.

Along with \setuphead ConTEXt provides the \setupheads command which globally affects all section commands. The ConTEXt wiki says, in reference to this command, that some people have said it does not work. According to my tests, this command works for some options but not others. In particular it does not work with the "style" option, which is striking, since the style for titles is most likely the one thing we would like to change globally so it affects all titles. But it does work, according to my tests, with other options such as, for example, "number" or "color". So, for example, \setupheads[color=blue] will ensure that all titles in our document are printed in blue.

Since I am a bit too lazy to bother testing every option to see if it works or not with \setupheads (remember that there are more than sixty of them) in what follows I will refer only to \setuphead.

Finally: before examining the specific options, we should note something that is said in ConTEXt wiki, although it is probably not said in the right place: some options only work if we are using the \start*SectionName* syntax.

> This information is contained in connection with \setupheads, but not \setuphead which is where the bulk of the options are explained and where, if it is only to be said in one place, it seems the most reasonable place to say it. On the other hand, the information only mentions the "insidesection" option, without making it clear whether or not it also happens with other options.

7.4.2 Parts of a section title

Before going into the specific options that allow us to configure the appearance of titles, it is advisable to start by pointing out that a section title can have up to three different parts, which ConTEXt allows us to format together or separately. These title elements are as follows:

- **The title itself**, meaning the text it is made up of. In principle this title is always displayed, except for sections of the "`part`" kind where the title is not displayed by default. The option that controls whether the title is displayed or not is "`placehead`" whose values can be "`yes`", "`no`" "`hidden`", "`empty`" or "`section`". The meaning of the first two is clear. But I am not so sure about the results of the remaining values of this option.

  Therefore, if we want the title to be displayed in the first level sections, our setting should be:

```
\setuphead
  [part]
  [placehead=yes]
```

  The title of certain sections, as we already know, can be automatically sent to headers and the table of contents. Using the `list` and `marking` options of section commands, we can indicate an alternative title to be sent instead. It is also possible, when writing the title, to use the `\nolist` or `\nomarking` commands to have certain parts of the title replaced by ellipses in the table of contents or header. For example:

```
\chapter{Influences of \nomarking{19th century} impressionism \nomarking{in the 21st century}}
```

  Will write "Influences of ... impressionism ..." in the header.

- **Numbering**. This is only the case for numbered sections (part, chapter, section, subsection...), but not for unnumbered ones (title, subject, subsubject). In fact, whether a particular section is numbered or not depends on the "`number`" and "`incrementnumber`" options whose possible values are "`yes`" and "`no`". In numbered sections both these are set as `yes` and in unnumbered sections, as `no`.

  Why are there two options to control the same thing? Because in fact the two options control two things; one is whether the section is numbered or not (`incrementnumber`) and the other is whether the number is displayed or not (`number`). If `incrementnumber=yes` and `number=no` are set for a section, we will get a section that is unnumbered outwardly (visually) but still counted internally. This would be useful for including such a section in

the table of contents, since ordinarily this would only include numbered sections. In this regard see subsection A in section 8.1.7.

- **The label** for the title. In principle this element in titles is empty. But we can associate a value with it, in which case, prior to the number and the actual title, the label we have assigned to this level will be printed. For example, in chapter titles we might want the word "Chapter" to be printed, or the word "Part" for parts. We do not use `\setuphead` to do this but the `\setuplabeltext` command.This command allows us to assign a textual value to the labels of the different sectioning levels. So, for example, if we want to write "Chapter" in our document before the chapter titles, we should set:

```
\setuplabeltext
  [chapter=Chapter~]
```

  In the example, after the assigned name, I have included the reserved "~" character that inserts an unbreakable blank space after the word. If we don't mind a line break happening between the label and the number, we could simply add a blank space. But this blank space (either kind) is important; without it the number will be connected to the label and we would see, for example, "Chapter1" instead of "Chapter 1".

7.4.3 Controlling the numbering (in numbered sections)

We already know that the predefined numbered sections (part, chapter, section...) and whether or not a particular section is numbered or not, depends on the "`number`" and "`incrementnumber`" options set up with `\setuphead`.

By default, the numbering of the various levels is automatic, unless we have assigned the value of "`yes`" to the "`ownnumber`" option. When "`ownumber=yes`" the number assigned to each command must be indicated. This is done:

- If the command is invoked using the classic syntax, by adding an argument with the number before the title text. For example:
  `\chapter{13}{Chapter title}` will generate a chapter that has manually been assigned the number 13.

- If the command has been invoked with the syntax specific to ConTEXt (`\SectionType [Options]` or `\startSectionType [Options]`), with the "`ownnumber`" option. For example:
  `\chapter[title={Chapter title}, ownnumber=13]`, will generate a chapter that has manually been assigned the number 13.

When ConTEXt automatically does the numbering, it uses internal counters that store the numbers of the different levels; thus there is a counter for parts, another for chapters, another for sections, etc. Each time ConTEXt finds a section command it carries out the following actions:

- It increases the counter associated with the level corresponding to that command by '1'.
- It resets the associated counters at all levels below that of the command in question to 0.

This means, for example, that each time a new chapter is found, the chapter counter is increased by 1 and all the section, subsection, subsubsection etc. commands are returned to 0; but the counter for parts is not affected.

To alter the number from which to start counting, use the `\setupheadnumber` command as follows:

```
\setupheadnumber[SectionType][Number from which to count]
```

where *Number from which to count* is the number from which sections of any type will be counted. So if *Number from which to count* is equal to zero, the first section will be 1; if it is equal to 10, the first section will be 11.

This command also allows us to alter the pattern for the automatic increment; so we can, for example, get the chapters or sections to be counted in pairs, or in threes. So, `\setupheadnumber[section][+5]` will see chapters numbered as 5 out of five; and `\setupheadnumber[chapter][14, +5]` will see that the first chapter begins with 15 (14+1), the second will be 20 (15+5), the third 25, etc.

By default, section numbering displays Arabic numbers, and the numbering of all previous levels is included. That is to say: in a document in which there are parts, chapters, sections and sub-sections, a specific sub-section will indicate to which part, chapter and section it corresponds. Thus the fourth sub-section of the second section of the third chapter of the first part will be "1.3.2.4".

The two basic options controlling how numbers are displayed are:

- **`conversion:`** It controls the type of numbering that will be used. It allows numerous values depending on the type of numbering we want:
  - **Numbering with Arabic numbers**: The classic numbering: 1, 2, 3, ... is obtained with the values `n, N` or `numbers`.
  - **Numbering with Roman numbers**. Three ways of doing this:
    - ⋆ Upper case Roman numbers: `I, R, Romannumerals`.
    - ⋆ Lower case Roman numbers: `i, r, Romannumerals`.

  - ⋆ Roman numbers in small caps: `KR, RK`.

- **Numbering with letters**. Three ways of doing this:

  - ⋆ Upper case letters: `A, Character`
  - ⋆ Lower case letters: `a, character`
  - ⋆ Letters in small caps: `AK, KA`

- **Numbering in words**. Meaning we write the word that designates the number. So, for example, '3' becomes 'Three'. There are two ways of doing this:

  - ⋆ Words beginning with a capital letter: `Words`.
  - ⋆ Words all in lower case: `words`.

- **Numbering with symbols**: Symbol-based numbering uses different sets of symbols in which each symbol is assigned a numerical value. As the symbol sets used by ConTEXt have a very limited number of them, it is only appropriate to use this type of numbering when the maximum number to be reached is not too high. ConTEXt provides for four different sets of symbols: `set 0, set 1, set 2 and set 3` respectively. Below are the symbols that each of these sets uses for numbering. Note that the maximum number that can be reached is 9 in `set 0` and `set 1` and 12 in `set 2` and `set 3`:

  Set 0: • – ⋆ ▷ ∘ ○ ◯ □ ✓
  Set 1: ⋆ ⋆⋆ ⋆⋆⋆ ‡ ‡‡ ‡‡‡ * ** ***
  Set 2: ⁎ † ‡ ** †† ‡‡ *** ††† ‡‡‡ **** †††† ‡‡‡‡
  Set 3: ⋆ ⋆⋆ ⋆⋆⋆ ‡ ‡‡ ‡‡‡ ¶ ¶¶ ¶¶¶ § §§ §§§

- `sectionsegments:` This option allows us to control whether or not to display the numbering for the preceding levels. We can indicate which previous levels will be displayed. This is done by identifying the initial level and the final level to be displayed. The identification of the level can be done by its number (part=1, chapter=2, section=3, etc.), or name (part, chapter, section, etc.). So, for example, "`sectionsegments=2:3`" indicates that chapter and section numbering should be displayed. It is exactly the same as saying "`sectionsegments=chapter:section`". If we want to indicate that all numbers above a certain level are displayed we can use, as a value of "`optionsegments`" *Initial Level:all*, or *InitialLevel:**. For example, "`sectionsegments=3:*`" indicates that numbering is displayed starting from level 3 (section).

So, for example, imagine that we want the parts to be numbered with Roman numerals in capital letters; the chapters with Arabic numerals, but without including

the number of the part to which they belong; the sections and subsections with Arabic numerals including the chapter and section numbers, and the subsections with capital letters. We should write the following:

```
\setuphead[part][conversion=I]
\setuphead[chapter]      [conversion=n, sectionsegments=2]
\setuphead[section]      [conversion=n, sectionsegments=2:3]
\setuphead[subsection]   [conversion=n, sectionsgments=2:4]
\setuphead[subsubsection][conversion=A, sectionsegments=5]
```

7.4.4 Title colour and style

We have the following options to control style and colour:

- **The style** is controlled with the "`style`", "`numberstyle`" and "`textstyle`" options depending on whether we want to affect the whole title, only the numbering, or only the text. By means of any of these options we can include commands that affect the font; namely: specific font, style (roman, sans serif or typewriter), alternative (italic, bold, slanted...) and size. If we only want to indicate one style feature we can do this either by using the name of the style (for example, "bold" for bold), or by indicating its abbreviation ("bf"), or the command that generates it (`\bf`, in the case of bold). If we want to indicate several features simultaneously, we must do it by means of the commands that generate them, writing them one after the other. Bear in mind, on the other hand, that if we indicate only one feature, the rest of the style features will be established automatically with the default values of the document, which is why it is rarely advisable to establish only one style feature.

- **The colour** is set with the "`color`", "`numbercolor`" and "`textcolor`" options depending on whether we want to set the colour of the whole title, or just the colour of the numbering or the text. The colour indicated here can be one of ConTEXt's predefined colours, or some other colour we have defined ourselves and previously assigned a name to. However, we cannot directly use a colour definition command here.

In addition to these six options, there are still five more options available for establishing some more sophisticated features with which we can do virtually anything we want. These are: "`command`", "`numbercommand`", "`textcommand`","`deepnumbercommand`" and "`deeptextcommand`". Let's begin by explaining the first three:

- **command** indicates a command that will take two arguments, the number and the section title. It can be a normal ConTEXt command or a command we have defined ourselves.
- **numbercommand** is similar to "command", but this command only takes an argument with the section number.
- **textcommand** is also similar to "command", but it only takes an argument with the title text.

These three options allow us to do practically anything we want. For example, if I want the sections to be right-aligned, enclosed in a frame and with a line break between the number and the text, I can simply create a command that does that, and then indicate that command as the value of the "command" command. This would be achieved with the following lines:

```
\define[2]\AlignSection
  {\framed[frame=on, width=broad,align=flushright]{#1\\#2}}

\setuphead
  [section]
  [command=\AlignSection]
```

When we simultaneously set the "command" and the "style" options, the command is applied to the title with its style. This means, for example, that if we we have set "textstyle=\em", and "textcommand=\WORD", the command \WORD (which capitalizes the text it takes as an argument) will be applied to the title with its style, i.e.: \WORD{\em Title text}.If we want it to be done the other way around, i.e. that the style is applied to the content of the title once the command has been applied, we should use, instead of the "textcommand" and "numbercommand" options, the "deeptextcommand" and "deepnumbercommand" options. This, in the example given above, would generate "{\em\WORD{Title text}}".

In most cases there would be no difference in doing it one way or the other. But in some cases there may be.

7.4.5 Location of number and title text

The "alternative" option controls two things simultaneously: the location of the numbering with respect to the title's text, and the location of the title itself (including number and text) with respect to the page on which it is displayed and the contents of the section. They are two different things, but as they are both governed by the same option, they are controlled simultaneously.

The location of the title in relation to the page and the first paragraph of the section content is controlled by the following possible values of "alternative":

- **text:** The section title is integrated with the first paragraph of its contents. The effect is similar to what is produced in LaTeX with \paragraph and \subparagraph.
- **paragraph:** The section title will be an independent paragraph.
- **normal:** The section title will be placed in the default location provided by ConTeXt for the particular type of section in question. Normally it is "paragraph".
- **middle:** The title is written as an autonomous, centred paragraph. If it is a numbered command, the number and the text are separated on different lines, both centred.

  An effect similar to what is obtained with "alternative=middle" is obtained with the "align" option that controls title alignment. It can take the values "left", "middle" or "flushright". But if we centre the title with this option, the number and the text will appear on the same line.
- **margintext:** This option causes the entire title (numbering and text) to be printed in the space reserved for the margin.

The location of the number in relation to the title text is indicated by the following possible values of "alternative":

- **margin/inmargin:** The title is a separate paragraph. The numbering is written in the space reserved for the margin. I haven't figured out the difference between using "margin" and using "inmargin".
- **reverse:** The title makes up a separate paragraph, but the normal order is reversed, and the text is printed first, then the number.
- **top/bottom:** In titles whose text occupies more than one line, these two options control whether the numbering will be aligned with the first line of the title or with the last line respectively.

7.4.6 Commands or actions to be carried out before or after printing the title

It is possible to indicate one or more commands that are executed before printing the title ("before" options) or after ("after" option). These options are widely used to visually mark the title. For example: if we want to add more vertical space between the title and the text that precedes it, "before=\blank" will add a blank line. To add even more space we could write "before={\blank[3*big]}". In this case we have surrounded the value of the option with curly brackets to avoid an error. We could also visually indicate the distance between the previous

text and the following one with "`before=\hairline, after=\hairline`", which would draw a horizontal line before and after the title.

Very similar to the "`before`" and "`after`" options are the "`commandbefore`" and "`commandafter`" ones. According to my tests I deduce that the difference is that the former two execute actions before and after starting to typeset the title as such, while the latter two refer to commands that will be executed before and after typesetting *the title text*.

If we want to insert a page break before the title, we have to use the "`page`" option that allows, among other values, "yes" for inserting a page break, "left" to insert as many page breaks as necessary to ensure that the title starts on an even page, "right" to ensure that the title starts on an odd page, or "no" if what we want is to disable the forced page break. This option, on the other hand, for levels below "`chapter`", will only work if the "`continue=no`" is used, otherwise it will not work if the section, subsection or command is on the first page of a chapter.

> By default, chapters start on a new page in ConTEXt. If it is established that the sections also start a new page, the problem arises of what to do with the first section of a chapter which, perhaps, is at the beginning of the chapter: if that section also starts a page break, we end up with the page which opens the chapter only containing the title of the chapter, which is not very aesthetic. This is why we can set the "`continue`" option, a name, I have to say, that is not very clear to me: if "`continue=yes`", the page break will not apply to the sections that are on the first page of a chapter. If "`continue=no`" the page break will still be applied.

If, instead of section commands we use section environments (`\start ... \stop`), we also have the "`insidesection`" option, by which we can indicate one or more commands that will be executed once the title has been typeset and we are already inside the section. This option would allow us, for example, to make sure that immediately after starting a chapter, a table of contents will be typeset automatically with ("`insidesection=\placecontent`")

7.4.7 Other configurable features

As well as those we have already seen, we can configure the following additional features with `\setuphead`:

- **Interlined**. Controlled by the "`interlinespace`" which takes as its value the name of an interline command previously created with `\defineinterlinespace` and configured with `\setupinterlinespace`.
- **Alignment**. The "`align`" option affects the alignment of the paragraph containing the title. Among others it can have the following values: "`flushleft`" (left), "`flushright`" (right), "`middle`" (centred), "`inner`" (inner margin) and "`outer`" (outer margin).
- **Margin**. With the "`margin`" option we can manually set the title's margin.

- **Indenting the first paragraph**. The value of the "`indentnext`" option (that can be "yes", "no" or "auto") controls whether or not the first line of the first paragraph of the section will be indented. Whether or not it should be indented (in a document where the first line of the paragraphs is generally indented) is a matter of taste.

- **Width**. By default, titles take up the width they need unless this is greater than the width of the line, in which case the title will take up more than one line. But with the "`width`" option we can assign a particular width for the title. The "`numberwidth`" and "`textwidth`" options respectively, assign the numbering width or the width of the title's text.

- **Separating number and text**. The "`distance`" and "`textdistance`" options allow us to control the distance separating the number from its text.

- Style of section headers and footers. For this we use the "`header`" and "`footer`" options

7.4.8 Other `\setuphead` options

With the options we already seen, we can see that the configuration possibilities for section titles are almost unlimited. However, `\setuphead` has around thirty options that I have not mentioned. Most because I have not discovered what they are for or how they are used, a few because their explanation would force me to go into aspects that I do not intend to deal with in this introduction.

7.5 Defining new section commands

We can define our own section commands with `\definehead` whose syntax is:

```
\definehead[CommandName][Model][Configuration]
```

where

- **CommandName** represents the name the new section command will have.

- **Model** is the name of an existing section command that will be used as a model from which the new command will initially inherit all its characteristics.

  In fact, the new command inherits much more than its initial characteristics from the model: it becomes a kind of customised instance of the model, but shares with it, for example, the internal counter that controls the numbering.

- **Configuration** is the customised configuration of our new command. Here we can use exactly the same options as in `\setuphead`.

It is not necessary to configure the new command at the time of its creation. This can be done later with \setuphead and, in fact, in the examples given in the ConTEXt manuals and its wiki, this seems to be the normal way.

7.6 The document's macrostructure

Chapters, sections, subsections, titles..., structure the document; they organise it. But together with the structure resulting from these kinds of commands, in certain printed books, especially those coming from the academic world, there is a kind of *macro-ordering* of the book's material, taking into account not its content but the function that each of these large parts performs in the book. This is how we differentiate between:

- The initial part of the document containing the title page, acknowledgement page, a dedication page, the table of contents, perhaps a preface, presentation page, etc.
- The main body of the document, which contains the fundamental text of the document, divided into parts, chapters, sections, subsections, etc. This part is usually the most extensive and important.
- Additional material made up of appendices or annexes that develop or exemplify some issue dealt with in the main body, or provide additional documentation not written by the author of the main body, etc.
- The final part of the document where we can find the bibliography, indexes, glossaries, etc.

In the source file we can demarcate each of these parts through the environments seen in table 7.2.

Part of the document	Command
Initial part	\startfrontmatter [Options] ... \stopfrontmatter
Main body	\startbodymatter [Options] ... \stopbodymatter
Appendices	\startappendices [Options] ... \stopappendices
Final part	\startbackmatter [Options] ... \stopbackmatter

Table 7.2 Environments that reflect the document's macrostructure

The four environments allow the same four options: "`page`", "`before`", "`after`" and "`number`", and their values and usefulness are the same as those found in \setuphead (see section 7.4), though we should note that here the "`number=no`" option will eliminate the numbering of all sectioning commands within the environment.

To include any of these large sections in our document only makes sense if it is to establish some kind of differentiation between them. Perhaps headers or page

numbering in frontmatter. Configuration of each of these blocks is achieved by `\setupsectionblock` whose syntax is:

```
\setupsectionblock[Block name] [Options]
```

where *Block name* can be `frontpart`, `bodypart`, `appendix` or `backpart` and the options can be the same as just mentioned: "`page`","`number`", "`before`" and "`after`". So, for example, to ensure that in *frontmatter* the pages are numbered with Roman numbers, in the preamble of our document we should write:

```
\setupsectionblock
  [frontpart]
  [
    before={\setuppagenumbering[conversion=Romannumerals]}
  ]
```

ConTEXt's default configuration for these four blocks implies that:

- The four blocks begin a new page.
- Section numbering changes in each of these blocks:
  - In `frontmatter` and `backmatter` by default all numbered sections are unnumbered.
  - In `bodymatter` chapters have Arabic numbering.
  - In `appendices` chapters are numbered with upper case letters.

It is also possible to create new section blocks with `\definesectionblock`.

Chapter 8
Table of contents, indexes, lists

A table of contents and an index are a global aspect of a document. Almost all documents will have a table of contents, while, only some documents will have an index. For many languages (but not for English) both the table of contents and the index come under the general term 'index'. For English readers, a table of contents will normally come at the beginning (of a document, or possibly in some cases at the beginning of chapters as well), and the index will come at the end.

Either of these imply a particular application of the mechanism for internal references whose explanation is included in section 9.2.

8.1 Table of contents

8.1.1 Overall view of the table of contents

In the previous chapter we examined the commands that allow the structure of a document to be established as it has been written. This section focuses on the table of contents and the index, which in some way *mirror* the document's structure. The table of contents is very useful for getting an idea of the document as a whole (it helps contextualise information) and for searching the exact point where a

particular passage might be located. Books with a very complex structure, with many sections and subsections with various levels of depth, seem to require a different kind of table of contents, since a poorly detailed one (perhaps with only the first two or three levels of sectioning) helps a lot to get an overall idea of the contents of the document, but is not very useful for locating a particular passage; unlike a very detailed table of contents, on the other hand, where it is easy to miss the forest for the trees and lose the overall view of the document. This is why, sometimes, books with a particularly complex structure include more than one table of contents: one not too detailed at the beginning showing the main parts, and a more detailed table of contents at the beginning of each chapter as well as, perhaps, an index at the end.

These can all be generated by ConTEXt automatically with relative ease. We can:

- Generate a complete or partial table of contents at any point in the document.
- Decide on the contents of either.
- configure their appearance down to the last detail.
- Include hyperlinks in the table of contents that allow us to jump directly to the section in question.

In fact this last utility is included by default in all tables of contents provided that the interactivity function has been enabled in the document. See, in this respect, section 9.3.

The explanation of this in the ConTEXt reference manual is, in my opinion, somewhat confusing, which I think is due to the fact that too much information is introduced at once. The mechanism for building ConTEXt tables of contents has many pieces to it; and it is difficult for a text that tries to explain them all at once to be clear. Especially for the reader who is new to the scene. By contrast, the explanation in wiki, is practically limited to examples: very useful for learning *tricks* but inadequate – I think – for understanding the mechanism and how it works. this is why the strategy I have decided to use to explain things in this introduction begins by assuming something that is not strictly true (or not completely true): that there is something in ConTEXt called the *table of contents*. Starting with this, the *normal* commands for generating the table of contents are explained, and when these commands and their configuration are well known, I think this is the moment for introducing – though at a theoretical rather than more practical level – the information on those pieces of the mechanism that have been omitted up till then. Knowledge of these additional *pieces* allows us to create much more customised tables of contents than the ones we can call the *normal ones* created with the commands explained up to that point; however, in most cases we will not need to do this.

8.1.2 Completely automatic table of contents with a title

The basic commands for generating an automatically generated table of contents (TOC) from the numbered sections of a document (`part`, `chapter`, `section`, etc.) are `\completecontent` and `\placecontent`. The main difference between the two commands is that the first adds a *title* to the TOC; to do so, immediately before the TOC it inserts an *unnumbered chapter* whose default title is Table of Contents.

Therefore `\completecontent`:

- Inserts, at the point where it is found, a new unnumbered chapter entitled "Table of Contents".

  > We recall that in ConTEXt the command used to generate an unnumbered section at the same level as chapters, is `\title` (see section 7.2). Therefore in reality `\completecontent` does not insert a *Chapter* (`\chapter`) but a *Title* (`\title`). I have not said so because I think it may be confusing for the reader to use the names of the unnumbered section commands here, since the term *Title* also has a broader sense, and it is easy for the reader not to identify it with the concrete level of sectioning we are referring to.

- This *chapter* (actually, `\title`) is formatted exactly the same as the rest of the unnumbered chapters in the document; which by default includes a page break.

- The table of contents is printed immediately after the title.

Initially the generated TOC is *complete*, as we can deduce from the command name that generates it (`\completecontent`). But on the one hand we can limit the level of depth of the TOC as explained in section 8.1.3 and, on the other, since this command is *sensitive* to the place it is found in the source file (see what is said further on about `\placecontent`), if `\completecontent` is not found at the beginning of the document it is possible that the TOC generated is not complete; and in some points of the source file it is even possible that the command is apparently ignored. If this happens, the solution is to invoke the command with the "`criterium=all`" option. Regarding this option, also see section 8.1.3.

To change the default title assigned to the TOCs we use the `\setupheadtext` command whose syntax is:

```
\setupheadtext[Language][Element=Name]
```

where *Language* is optional and refers to the language identifier used by ConTEXt (see section 10.5), and *Element* refers to the element whose name we want to change ("`content`" in the case of the table of contents) and *Name* is the name or title we want to give our TOC. For example

```
\setupheadtext[en][content=Contents]
```

will ensure that the TOC generated by \completecontent is entitled “Contents” instead of “Table of Contents”.

Moreover, \completecontent allows the same configuration options as \placecontent, for the explanation of which I refer to (the next section).

8.1.3 Automatic table of contents without a title

The general command for inserting a TOC without a title, generated automatically from the document's sectioning commands, is \placecontent, whose syntax is:

```
\placecontent[Options]
```

In principal, the table of contents will contain absolutely all numbered sections, although we can limit its level of depth with the \setupcombinedlist command (that we will speak of further on). So, for example:

```
\setupcombinedlist[content][list={chapter,section}]
```

will limit the contents of the TOC to chapters and sections.

A peculiarity of this command is that it is sensitive to its location in the source file. This is very easy to explain with a few examples, but much more difficult if we want to specify exactly how the command works and which headings are included in the TOC in each case. So let's start with the examples:

- \placecontent placed at the beginning of the document, before the first section command (part, chapter or section, according to the situation) will generate a complete table of contents.

  > I am not really sure that the table of contents generated by default is *complete*, I believe it does include enough levels of sectioning to be complete in most cases; but I suspect it will not go beyond the eighth level of sectioning. In any case, as mentioned above, we can adjust the sectioning level the TOC reaches with
  >
  > ```
  > \setupcombinedlist[content][list={chapter, section, subsection, ...}]
  > ```

- By contrast, this same command located inside a part, chapter or section will exclusively generate a TOC of the content of that element, or in other words chapters, sections and other lower levels of sectioning of a specific part, or sections (and other levels) of a specific chapter, or subsections of a specific section.

As for the technical and detailed explanation, in order to understand the default operation of \placecontent properly, it is essential to remember that the various sections are, in fact, *environments* for ConTEXt Mark IV that start with

\startSectionType and end with \stopSectionType and can be contained within other lower level section commands. So, taking that into account, we can say that \placecontent generates by default a table of contents that will only include:

- Elements that belong to the *environment* (section level) where the command is placed. This means that the command when placed in a chapter will not include sections or subsections from other chapters.
- Elements that have a sectioning level lower than the level corresponding to the point where the command is located. Meaning that if the command is in a chapter, only sections, subsections and other lower levels are included; but if the command is in a section, it will be split to make the TOC of the subsection level.

Furthermore, for the table of contents to be generated, it is required that \placecontent be found *before* the first section of the chapter in which it is located, or before the first subsection of the section in which it is located, etc.

I'm not sure I was clear in the explanation above. Perhaps with a somewhat more detailed example than the previous ones we can better understand what I mean: let's imagine the following structure of a document:

- Chapter 1
  - Section 1.1
  - Section 1.2
    - Subsection 1.2.1
    - Subsection 1.2.2
    - Subsection 1.2.3
  - Section 1.3
  - Section 1.4
- Chapter 2

So: \placecontent placed before Chapter 1 will generate a complete table of contents, similar to the one generated by \completecontent but without a title. But if the command is placed within Chapter 1 and before section 1.1, the table of contents will be only of the chapter; and if it is placed at the beginning of section 1.2, the table of contents will be only the content of that section. But if the command is placed, for example, between sections 1.1 and 1.2 it will be ignored. It will also be ignored if it is placed at the end of a section, or at the end of the document.

All of this, of course, refers only to the case where the command does not include options. In particular, the `criterium` option will alter that default behaviour.

Of the options allowed by \placecontent I will only explain two of them, the most important ones for setting up the TOC, and, moreover, the only ones that are (partially) documented in the ConTEXt reference manual. The `criterium` option, which affects the content of the TOC in relation to the place in the source file

where the command is located; and the `alternative` option, which affects the general layout of the TOC to be generated.

8.1.4 Elements to incorporate in the TOC: the `criterium` option

The default operation of `\placecontent` in relation to the position of the command in the source file has been explained above. The `criterium` option alters this operation. Among others, it can take the following values:

- `all`: the TOC will be complete, regardless of the place in the source file where the command is found.
- `previous`: the TOC will only include the section commands (of the level we are at) *previous* to `\placecontent`. This option is intended for TOCS that are written at the end of the document or section in question.
- `part, chapter, section, subsection...`: implies that the TOC should be limited to the sectioning level indicated.
- `component`: in multifile projects (see section 4.6), it will generate only the TOC corresponding to the *component* where the `\placecontent` or `\completecontent` command is found.

8.1.5 Layout of the table of contents: the `alternative` option

The `alternative` option controls the overall layout of the table of contents. Its main values can be see in table 8.1.

alternative	Contents of TOC entries	Notes
a	Number – Title – Page	One line per entry
b	Number – Title – Spaces – Page	One line per entry
c	Number – Title – Leader dots – Page	One line per entry
d	Number – Title – Page	Continuous TOC
e	Title	Framed
f	Title	Left aligned, right aligned or centred
g	Title	Centred

Table 8.1 Ways of formatting the table of contents

The first four alternative values provide all the information of each section (its number, its title and the page number where it begins), and are therefore suitable for both paper and electronic documents. The last three alternatives only inform us about the title, so they are only suitable for electronic documents where it is

not necessary to know the page number where a section begins, provided that the TOC includes a hyperlink to it, which happens by default in ConTEXt.

Furthermore, I believe that in order to truly appreciate the differences between the various alternatives, it is best for the reader to generate a test document where he or she can analyse them in detail.

8.1.6 Format of TOC entries

We have seen that the `alternative` option of `\placecontent` or `\completecontent` allows us to control the general *layout* of the table of contents, i.e. what information will be shown for each heading, and whether or not there will be line breaks separating the different headings. Final adjustments to each TOC entry are made with the `\setuplist` command whose syntax is as follows:

```
\setuplist[Element][Configuration]
```

where *Element* refers to a particular kind of section. This could be `part`, `chapter`, `section`, etc. We can also configure more than one element at the same time, separating them with commas. *Configuration* has up to 54 possibilities, many of them, as usual, not expressly documented; but this does not prevent those that are documented, or the ones that are not clear enough from allowing full adjustment of the TOC.

I will now explain the most important options, grouping them according to their usefulness, but before going into them let us remember that a TOC entry, depending on the value of the `alternative`, can have up to three different components: The section number, the title of the section, and the page number. The configuration options allow us to configure the various components globally or separately:

- *Inclusion (or not) of the different components*: If we have chosen an alternative that includes, in addition to the title, the section number and the page number (alternatives 'a' 'b' 'c' or 'd'), the options `headnumber=no` or `pagenumber=no`, it means that for the specific level we are configuring, the section number (`headnumber`) or the page number (`pagenumber`) is not displayed.

- *Colour and style*: We already know that the entry that generates a specific section in the TOC may have (depending on the alternative) up to three different components: section number, title and page number. We can jointly indicate the style and the colour for the three components using the `style` and `color` options, or do it individually for each component by means of `numberstyle, textstyle` or `pagestyle` (for the style) and `numbercolor, textcolor` or `pagecolor` for the colour.

  To control the appearance of each entry, in addition to the style itself, we can apply some command to the whole entry or to one of its different elements. For

this there are the `command`, `numbercommand`, `pagecommand` and `textcommand` options. The command indicated here can be a standard ConTEXt command or a command of our own creation. Section number, title text and page number will be passed as arguments to the `command` option, while the section title will be passed as an argument to `textcommand` and page number to `pagecommand`. So, for example, the following sentence will ensure that section titles are written in (fake) small caps:

```
\setuplist[section][textcommand=\Cap]
```

- *Separation of the other TOC elements*: The `before` and `after` options allow us to indicate the commands that will be executed before (`before`) and after (`after`) typesetting the TOC entry. Normally these commands are used to set either the spacing or some separating element between the previous and subsequent entries.

- *Indenting an element*: set with the `margin` option which allows us to set the amount of left indentation that the entries of the level we are configuring will have.

- *Hyperlinks embedded in the TOC*: By default the index entries include a hyperlink to the document page where the section in question begins. Using the `interaction` option we can disable this function (`interaction=no`) or limit the part of the index entry where the hyperlink will be, which can be the section number (`interaction=number` or `interaction=sectionnumber`), the section title (`interaction=text` or `interaction=title`) or the page number (`interaction=page` or `interaction=pagenumber`).

- *Other aspects*:

  – `width`: specifies the separation distance between the number and title of the section. It can be a dimension, or the keyword `fit` that sets the exact width of the section number.

  – `symbol`: allows the section number to be replaced by a *symbol*. Three possible values are supported: `one`, `two` and `three`. The value `none` for this option removes the section number from the TOC.

  – `numberalign`: indicates the alignment of numbering elements; it can be left, right, middle, flushright, flushleft.

Among the multiple configuration options of the TOC, there are none that allows us to directly control the interline spacing. This will be, by default, the one that applies to the document as a whole. Often, however, it is preferable that lines in the TOC are slightly *tighter* than the rest of the document. To achieve this we should enclose the command that generates the table of contents (`\placecontent`

or \completecontent) within of a group where a different interline spacing is established. For example:

```
\start
  \setupinterlinespace[small]
  \placecontent
\stop
```

8.1.7 Manual adjustments to the table of contents

We have already explained the two fundamental commands for generating tables of contents (\placecontent and \completecontent), as well as their options. With these two commands, TOCs are automatically generated, constructed from the existing numbered sections in the document, or in the block or segment of the document to which the table of contents refers. I will now explain certain *settings* that we can make so that the content of the TOC is not so *automatic*. This implies:

- The possibility of also including some unnumbered section titles in the TOC.
- The possibility of manually sending a particular entry to the TOC that does not correspond to the presence of a numbered section.
- The possibility of excluding a particular numbered section from the TOC.
- The possibility that the title for a particular section reflected in the TOC does not coincide exactly with the title included in the body of the document.

A. Including unnumbered sections in the TOC

The mechanism by which ConTEXt builds the TOC means that all numbered sections are automatically included, which, as I have already said (see section 7.4.2) depends on the two (`number` and `incrementnumber`) options that we can change with \setuphead for each kind of section. It was also explained there that a section type where `incrementnumber=yes` and `number=no` would be an internally but not externally numbered section.

Therefore, if we want a particular unnumbered section type – for example, `title` – to be included in the TOC, we must change the value of the `incrementnumber` option for that section type, setting it to `yes` and then include that section type among those to be displayed in the TOC, which is done, as explained above, with \setupcombinedlist:

```
  \setuphead
    [title]
    [incrementnumber=yes]

\setupcombinedlist
  [content]
```

```
[list={chapter, title, section, subsection, subsubsection}]
```

We can then, if we wish, format this entry using `\setuplist` in exactly the same way as any of the others; for example:

```
\setuplist[title][style=bold]
```

Note: The procedure just explained will include all instances in our document of the unnumbered section type concerned (in our example the `title` type sections). If we only wish to include a particular occurrence of that section type in the TOC, it is preferable to do so by the procedure explained below.

B. Manually adding entries to the TOC

We can send either an entry (simulating the existence of a section that does not really exist) or a command to the table of contents, from any point in the source file.

To send an entry that simulates the existence of a section that does not really exist, use the `\writetolist` whose syntax is:

```
\writetolist[SectionType][Options]{Number}{Text}
```

in which

- The first argument indicates the level that this section entry must have in the TOC: `chapter`, `section`, `subsection`, etc.
- The second argument, which is optional, allows this entry to be configured in a particular way. If the manually sent input is omitted, it will be formatted as are all the entries of the level indicated with the first argument; although, I must point out that in my tests I have not managed to make it work.

Both in the official list of ConTEXt commands (see section 3.6) and in the wiki we are told that this argument allows the same values as `\setuplist` which is the command that allows us to format the different TOC entries. But, I insist, in my tests I have not managed to change the appearance of the TOC entry sent manually in any way.

- The third argument is supposed to reflect the numbering that the element sent to the TOC has, but I couldn't get this to work in my tests either.
- The last argument includes the text to be sent to the TOC.

This is useful, for example, if we want to send a particular unnumbered section, but only that to the TOC. In section A it explains how to get an entire category of unnumbered sections to be sent to the table of contents; but if we only want to send a particular occurrence of a section type to it, it is more convenient to use the `\writetolist` command. And so, for example, if we want the section of our

document containing the bibliography not to be a numbered section, but still to be included in the TOC, we would write:

```
\subject{Bibliography}
\writetolist[section]{}{Bibliography}
```

See how we are using the unnumbered version of `section`, which is `subject`, for the section but we are sending it to the index, manually, as if it were a numbered section (`section`).

Another command intended to influence the table of contents manually is `\writebetweenlist` which is used to send not an entry itself, but a *command* to the table of contents, from a particular point in the document. For example, if we want to include a line between two items in the TOC, we could write the following at any point in the document located between the two sections concerned:

```
\writebetweenlist[section]{\hrule}
```

C. Exclude a particular section from the TOC belonging to a section type that is included in the TOC

The table of contents is constructed from *section types* established, as we already know, by the `list` option of `\setupcombinedlist`, so if a certain *section type* must appear in the TOC, there is no way of excluding a particular section from it that for whatever reasons we don't want in the TOC.

Normally, if we don't want a section to appear there, what we would do is to use its *unnumbered equivalent* meaning, for example, `title` instead of `chapter`, `subject` instead of `section`, etc. These sections are not sent to the TOC, and neither are they numbered.

However, if for any reason we want a certain section to be numbered but not appear in the table of contents, even if other types of this kind do, we can use a *trick* which consists of creating a new section type that is a clone of the section in question. For example:

```
\definehead[MySubsection][subsection]
\section{First section}
\subsection{First subsection}
\MySubsection{Second subsection}
\subsection{Third subsection}
```

This will ensure that when inserting a section type `MySubsection` the subsection counter will increase, since this section is a *clone* of the subsections, but the TOC will not be altered, since by default it does not include `MySubsection` types.

D. Section title text which differs in the TOC from the title in the body of the document

If we do not want the title of a particular section included in the TOC to be identical to the one displayed in the body of the document, we have two procedures available to us:

- Creating the section not with traditional syntax (`SectionType{Title}`) but with `\SectionType [Options]`, or with `\startSectionType [Options]`, and assign the text we want to be written in the TOC to the `list` option (see section 7.3).
- When writing the title of the section in question in the body of the document, use the `\nolist` command: this command causes the text it takes as an argument to be replaced in the TOC by an ellipsis. For example:

```
\chapter
  [title={An \nolist{approximate and slightly repetitive}
          introduction to the reality of the obvious}]
```

would typeset as the chapter title in the body of the document, "An approximate and slightly repetitive introduction to the reality of the obvious", but would send the following text to the TOC "An ... introduction to the reality of the obvious".

> **Attention:** What I have just pointed out about the `\nolist` command is stated in both the ConTEXt reference manual and the wiki. For me, however, it produces a compiling error, telling me that the `\nolist` command is undefined.

8.2 Lists, combined lists and table of contents based on a list

Internally, for ConTEXt, a table of contents is nothing more than a *combined list*, which, in turn, as its name suggests, consists of a combination of simple lists. Therefore the basic notion from which ConTEXt builds the table of contents is that of a list. Several lists are combined to form a table of contents. By default, ConTEXt contains a predefined combined list called "`content`" and this is what the commands examined so far work with: `\placecontent` and `\completecontent`.

8.2.1 Lists in ConTEXt

In ConTEXt, a *list* is a range of numbered elements about which we need to remember three things:

1. The number.
2. The name or title.
3. The page where it is found.

This happens with numbered sections; but also with other elements of the document such as images, tables, etc. In general, those elements for which there is a command whose name begins with `\place` which places them as `\placetable`, `\placefigure`, etc.

In all these cases, ConTEXt automatically generates a list of the different times the type of element in question appears, its number, title and page. Thus, for example, there is a list of chapters, called `chapter`, another of sections, called `section`; but also another of tables (called `table`) or images (called `figure`). Lists generated automatically by ConTEXt are always called the same as the item they store.

A list will also be automatically generated if we create, for example, a new type of numbered section: when we create it we will be implicitly creating also the list that stores them. And if for a non-numbered section by default, we set the option `incrementnumber=yes`, making it a numbered section, we will also be implicitly creating a list with that name.

Together with the implicit lists (automatically defined by ConTEXt) we can create our own lists with `\definelist`, whose syntax is

```
\definelist[ListName][Configuration]
```

Items on the list are added:

- In lists predefined by ConTEXt, or created by it as a result of creating a new floating object (see section 13.5), automatically each time an item from the list is inserted into the document, either by a sectioning command or by the `\placeWhatever` command for other types of lists, for example: `\placefigure`, will insert any image in the document, but it will also insert the corresponding entry in the list.
- Manually in any kind of list with `\writetolist[ListName]`, already explained in subsection B of section 8.1.7. The `\writebetweenlist` command is also available. It too was explained in that section.

Once a list has been created and all its items included in it, the three basic commands related to it are `\setuplist`, `\placelist` and `\completelist`. The first allows us to configure what the list looks like; the last two insert the list in question at the point in the document where it finds them. The difference between `\placelist` and `\completelist` is similar to the difference between `\placecontent` and `\completecontent` (see sections 8.1.2 and 8.1.3).

So, for example,

```
\placelist[section]
```

will insert a list of the sections, including a hyperlink to them if the document's interactivity is enabled and if, in `\setuplist`, we have not set `interaction=no`. A list of sections is not exactly the same as a table of contents based on sections: the idea of a table of contents usually includes the lower levels as well (sub-sections, subsubsections, etc.). But a list of sections will include only the sections themselves.

The syntax of these commands is:

```
\placelist[ListName][Options]
```

```
\setuplist[ListName][Configuration]
```

The `\setuplist` options have already been explained in section 8.1.6, and the options for `\placelist` are the same as for `\placecontent` (see section 8.1.3).

8.2.2 Lists or indexes of images, tables and other items

From what has been said so far, it can be seen that, since ConTEXt automatically creates a list of images placed in a document with the `\placefigure` command, generating a list or index of images at a particular point in our document is as simple as using the `\placelist[figure]` command. And if we want to generate a list with a title (similar to what we get with `\completecontent`) we can do it with `\completelist[figure]`. We can do similarly with the other four predefined kinds of floating objects in ConTEXt: tables ("`table`"), Graphics ("`graphic`"), *intermezzos* ("`intermezzo`") and chemical formulas ("`chemical`"), although for specific cases of these, ConTEXt already includes a command that generates them without a title: (`\placelistoffigures`, `\placelistoftables`, `\placelistofgraphics`, `\placelistofintermezzi` and `\placelistofchemicals`), and another that generates them with a title: (`\completelistoffigures`, `\completelistoftables`, `\completelistofgraphics`, `\completelistofintermezzi` and `\completelistofchemicals`), in a similar way to `\completecontent`.

In the same way, for floating objects we ourselves have created (see section 13.5) the `\placelistof<FloatName>` and `\completelistof<FloatName>` will be automatically created.

For lists we have created with `\definelist` we can create an index with `\placelist[ListName]` or with `\completelist[ListName]`.

8.2.3 Combined lists

A combined list is, as its name suggests, a list that combines items from different previously defined lists. By default, ConTEXt defines a combined list for tables of content whose name is "`content`", but we can create other combined lists with `\definecombinedlist` whose syntax is:

```
\definecombinedlist[Name][Lists][Options]
```

where

- *Name*: is the name the new combined list will have.
- *Lists*: refers to the names of lists to be combined, separated by commas.
- *Options*: Configuration options for the list. They can be indicated at the time of defining the list, or, probably preferably, when the list is invoked. The main options (which have already been explained) are `criterium` (subsection 8.1.4 of section 8.1.3) and `alternative` (in subsection 8.1.5 in the same section).

A collateral effect of creating a combined list with `\definecombinedlist` is that it also creates a command called `\placeListName` which serves to invoke the list, that is: to include it in the output file. So for example,

```
definecombinedlist[TOC]
```

will create the command `\placeTOC`; and

```
definecombinedlist[content]
```

will create the command `\placecontent`

But wait, `\placecontent`! Isn't this the command that is used to create a *normal* table of contents? Indeed: this means that the standard table of contents is actually created by ConTEXt by means of the following command:

```
\definecombinedlist
  [content]
  [part, chapter, section, subsection,
    subsubsection, subsubsubsection,
    subsubsubsubsection]
```

Once our combined list is defined, we can configure it (or reconfigure it) with `\setupcombinedlist` which allows the already explained options `criterium` (see subsection 8.1.4 in section 8.1.3) and `alternative` (see subsection 8.1.5 in the same section), as well as the `list` option to *change* the lists included in the combined list.

The official list of ConTEXt commands (see section 3.6) does not mention the `list` option among the options allowed for `\setupcombinedlist`, but it is used in several examples of the use of this command in the wiki (which, moreover, does not mention it in the page devoted to this command either). I have also checked that the option works.

8.3 Index

8.3.1 Generating the index

A subject index consists of a list of significant terms, usually located at the end of a document, indicating the pages where such a subject can be found.

When books were put typeset by hand, generating a subject index was a complex task, as well as a tedious one. Any change in the pagination could affect all the entries in the index. Therefore, they were not very common. Today, the computer mechanisms for typesetting mean that, while the task is likely to continue being tedious, it is no longer so complex given that it is not so difficult for a computer system to maintain an up-to-date list of data associated with an index entry.

To generate a subject index we need:

1. Determine which words, terms or concepts are to be part of it. This is a task that only the author can do.

2. Check at which points in the document each entry in the future index appears. Although, to be precise, more than *checking* the places in the source file where the concept or issue is discussed, what we do when we work with ConTEXt is *to mark* those spots, inserting a command that will then serve to generate the index automatically. This is the tedious part.

3. Finally, we generate and format the index by placing it at the point of our choice in the document. The latter is quite simple with ConTEXt and requires only one command: `\placeindex`.

A. The prior definition of the entries in the index and the marking of the points in the source file that refer to them

The fundamental work is in the second step. It is true that computer systems also facilitate it in the sense that we can do a global text search to locate the places in the source file where a specific subject is treated. But we should also not blindly rely on such text searches: a good subject index must be able to detect every spot where a particular subject is being discussed, even if this is done without using the *standard* term to refer to it.

To *mark* an actual point in the source file, associating it with a word, term or idea that will appear in the index, we use the `\index` command whose syntax is as follows:

```
\index[Alphabetical]{Index entry}
```

where *Alphabetical* is an optional argument that is used to indicate an alternative text to that of the index entry itself in order to sort it alphabetically, and *Index entry* is the text that will appear in the index, associated with this mark. We can also apply the formatting features that we wish to use, and if reserved characters appear in the text, they must be written in the usual way in ConTEXt.

> The possibility of alphabetising an index entry in a way different from how it is actually written, is very useful. Think, for example, of this document, if I want to generate an entry in the index for all references to the `\TeX` command. For example, the sequence `\index{\backslash TeX}` will list the command not by the 't' in 'TeX', but among the symbols, since the term sent to the index begins with a backslash. This is done by writing `\index[tex]{\backslash TeX}`.

The *index entries* will be the ones we want. For a subject index to be really useful we have to work a little harder at asking what concepts the reader of a document is most likely to look for; so, for example, it may be better to define an entry as "disease, Hodgkins" than defining it as "Hodgkin's disease", since the more inclusive term is "disease".

> By convention, entries in a subject index are always written in lower case, unless they are proper names.

If the index has several levels of depth (up to three are allowed) to associate a particular index entry with a specific level the '+' character is used. As follows:

```
\index{Entry 1+Entry 2}
\index{Entry 1+Entry 2+Entry 3}
```

In the first case we defined a second level entry called *Entry 2* that will be a sub-entry of *Entry 1*. In the second case we defined a third level entry called *Entry 3* that will be a sub-entry of *Entry 2*, which in turn is a sub-entry of *Entry 1*. For example

```
My \index{dog}dog, is a \index{dog+greyhound}greyhound called Rocket.
He does not like \index{cat+stray}stray cats.
```

It is worth noting some details of the above:

- The `\index` command is usually placed *before* the word it is associated with and is normally not separated from it by a a blank space. This is to ensure that the command is on the exact same page as the word it is linked to:

  - If there were a space separating them, there could be the possibility that ConTEXt would choose just that space for a line break which could also end up being a page break, in which case the command would be on one page and the word it is associated with on the next page.

  - If the command were to come *after* the word, it would be possible for this word to be broken by syllables and a line break inserted between two of its syllables that would also be a page break, in which case the command would be pointing to the next page beginning with the word it points to.

- See how second level terms are introduced in the second and third appearances of the command.

- Also check how, in the third use of the `\index` command, although the word that appears in the text is "cats", the term that will be sent to the index is "cat".

- Finally: see how three entries for the subject index have been written in just two lines. I said before that marking the precise places in the source file is tedious. I will now add that marking too many of them is counter-productive. Too extensive an index is by no means preferable to a more concise one in which all the information is relevant. That is why I said before that deciding which words will generate entry in the index should be the result of a conscious decision by the author.

If we want our index to be truly useful, terms that are used as synonyms must be grouped in the index under one head term. But since it is possible for the reader to search the index for information by any of the other head terms, it is common for the index to contain entries that refer to other entries. For example, the subject index of a civil law manual could just as easily be something like

contractual invalidity
    see *nullity*.

We achieve this not with the `\index` command but with `\seeindex` whose format is:

```
\seeindex[Alphabetical] {Entry1} {Entry2}
```

where *Entry1* is the index entry that will refer to the other; and *Entry2* is the reference target. In our previous example we would have to write:

```
\seeindex{contractual invalidity}{nullity}
```

In \seeindex we can also use the '+' sign to indicate sub-levels for either of its two arguments in square brackets.

B. Generating the final index

Once we have marked all the entries for the index in our source file, the actual generation of the index is carried out using the \placeindex or \completindex commands. These two commands scan the source file for the \index commands, and generate a list of all the entries that the index should have, associating a term with the page number corresponding to where it found the \index command. Then they alphabetically order the list of terms that appear in the index and merge cases where the same term appears more than once, and finally, they insert the correctly formatted result in the final document.

The difference between \placeindex and \completeindex is similar to the difference between \content and \completecontent (see section 8.1.2): \placeindex is limited to generating the index and inserting it, while \completeindex previously inserts a new chapter in the final document, called "Index" by default, inside which the index will be typeset.

8.3.2 Formatting the subject index

Subject indexes are a particular application of a more general structure ConTEXt calls "*register*"; therefore the index is formatted with the command:

```
\setupregister[index][Configuration]
```

With this command we can:

- Determine what the index will look like with its different elements. Namely:
  - The index headings which are usually letters of the alphabet. By default these are in lower case. With `alternative=A` we can set them to be in upper case.
  - The entries themselves, and their page number. The appearance depends on the `textstyle, textcolor, textcommand` and `deeptextcommand` options for the actual entry, and `pagestyle, pagecolor` and `pagecommand`, for the page number. With `pagenumber=no` we can also generate a subject index without page numbers (although I don't know if this could be useful).
  - The `distance` option measures the width of separation between the name of an entry and the page numbers; but it also measures the amount of indentation for subentries.

The names of the `style`, `textstyle`, `pagestyle`, `color`, `textcolor`, and `pagecolor` options are clear enough to tell us what each one does I think. For `command`, `pagecommand`, `textcommand` and `deeptextcommand`, I refer to the explanation for similarly named options in section 7.4.4, regarding the configuration of section commands.

- To set the general appearance of the index, which includes, among others the commands to execute before (`before`) or after(`after`) the index, the number of columns it needs to have (`n`), whether the columns should be equal or not (`balance`), the alignment of entries (`align`), etc.

8.3.3 Creating other indexes

I have explained the subject index as if only one such index would be possible in a document; but the truth is that documents can have as many indexes as desired. There could be an index of personal names, for example, which collects the names of people mentioned in the document, with an indication of the place where they are cited. These are still a kind of index. In a legal text we could also create a special index for mentions of the Civil Code; or, in a document like the present one, an index of macros explained in it, etc.

To create an additional index in our document we use the `\defineregister` command whose syntax is:

```
\defineregister[IndexName] [Configuration]
```

where *IndexName* is the name the new index will have, and *Configuration* controls how it works. It is also possible to configure the index later on by means of

```
\setupregister[IndexName] [Configuration]
```

Once a newly named index *IndexName* has been created we will have the `\IndexName` command at our disposal to mark the entries that this index will have in a similar way to the way entries are marked with `\index`. The seeIndexName command also lets us create entries that refer to other entries.

For example: we could create an index of ConTEXt commands in this document with the command:

```
\defineregister[macro]
```

that would create the `\macro` command. This lets me mark all the references to ConTEXt commands as an index entry, and then generate the index with `\placemacro` or `\completemacro`.

> Creating a new index enables the `\IndexName` command to mark it entries, and the `\placeIndexName` and `\completeIndexName` commands for generating the index. But these latter two commands are actually abbreviations of two more general commands applied to the index in question. Thus, `\placeIndexName` is equivalent to `\placeregister[IndexName]` and `\completeIndexName` is equivalent to `\completeregister[IndexName]`.

Chapter 9
References and hyperlinks

9.1 Reference types

Scientific and technical documents abound in references:

- Sometimes they refer to other documents that are the basis for what is being said, or that contradict what is being explained, or that develop or further nuance the idea being dealt with, etc. In these cases the reference is said to be *external* and, if the document is to be academically rigorous, the reference takes the form of *citations* from the literature.

- But it is also common for a document, in one of its sections, to refer to another of its sections, in which case the reference is said to be *internal*. There is also an internal reference when a point in the document comments on some aspect of a particular image, table, note, or element of a similar nature, referring to it by its number or by the page on which it is found.

  For the purposes of precision, internal references need to be aimed at an exact and easily identifiable place in the document. Hence these kinds of references are always a reference to either numbered elements (as, for example, when we say “see table 3.2”, or “Chapter 7”), or page numbers. Vague references of the “as we have already said” or “as we will see further on” kind are not true references, and there is no special requirement for typesetting them, nor is there any special tool for doing so. Also, I personally dissuade my PhD or MA students from any habitual use of this practice.

> Internal references are also commonly called "cross references" though in this document I will simply use the term "references" in general, and "internal references" when I wish to be specific.

In order to clarify the terminology I am using for references, I will call the point in the document where a reference is introduced the *origin*, and the location to which it points, the *target*. Seen this way, we would say that a reference is an internal one when the origin and target are in the same document, and an external one when origin and target are in different documents.

From the point of view of typesetting the document:

- External references pose no special problem and therefore, in principle, do not require any tool to introduce them: all the data I need from the target document are available to me and I can use them in the reference. However, if the document of origin is an electronic document and the target document is also available on the Web, then it is possible to include a hyperlink in the reference that allows one to jump directly to the target. In these cases the document of origin can be said to be *interactive*.

- By contrast, internal references do pose a challenge for typesetting the document, since anyone who has experience in the preparation of moderately long scientific and technical documents knows that it is almost inevitable that numbering of pages, sections, images, tables, theorems or similar to what is indicated in the reference, will change during the document's preparation, which makes it very difficult to keep it up to date.

> In pre-computer times, authors avoided internal references; and those that were inevitable, such as the table of contents (which, if accompanied by the page number of each section, is an example of an internal reference), were written at the end.

Even the most limited typesetting systems, such as word processors, allow for the inclusion of some kind of internal cross-references such as tables of contents. But that is nothing compared to the comprehensive reference management mechanism included in ConTEXt, which can also combine the internal reference management mechanism aimed at keeping references up to date, with the use of hyperlinks which is obviously not exclusive to external references.

9.2 Internal references

Two things are needed to establish an internal reference:

1. A label or identifier at the target point. While compiling, ConTEXt, will associate particular data with this label. What data will be associated depends on the kind of label it is; it can be the section number, the note number, the

image number, the number associated with a particular item in a numbered list, the section title, etc.

2. A command at the point of origin that reads the data associated with the label linked to the target point and inserts it at the point of origin. The command varies depending on which data from the label we want to insert at the point of origin.

When we think about a reference, we do so in terms of "origin ⟶ target", so it might seem that matters relating to the origin should be explained first, and then those relating to target. However, I believe that it is easier to understand the logic of references if the explanation is reversed.

9.2.1 The label in the reference target

In this chapter, by *label* I mean a text string that will be associated with the target point of a reference and used internally to retrieve certain information regarding the target point of a reference such as, for example, page number, section number etc. In fact, the information associated with each label depends on the procedure for creating it. ConTEXt calls these labels *references*, but I think that this latter term, as it has a much broader meaning, is less clear.

The label associated with the target reference:

- Needs each potential target in the document to be a unique one so it can be identified without doubt. If we use the same label for different targets, ConTEXt will not throw a compiling error but it will cause all references to point to the first label it finds (in the source file) and this will have the side effect that some of our references may be wrong, and, worse still, that we do not notice them. Therefore, it is important to make sure, when creating a label, that the new label we are assigning has not already been assigned before.

- It can contain letters, digits, punctuation marks, blank spaces, etc. Where there happen to be blank spaces, ConTEXt's general rules regarding these kinds of characters still apply (see section 4.2.1), so that, for example, "`My nice label`" and "`My  nice  label`" are seen as the same, even though a different number of blank spaces is used in both.

Since there is no limitation as to which characters can be part of the label and how many there are, my advice is to use label names that are clear, and will help us to understand the source file when, perhaps, we read it long after it was originally written. That's why the example I gave before ("My nice label") is not a good example, as it does not tell us anything about the target the label is pointing to. For this heading, for example, the label 'sec:Target labels' would be better.

To associate a particular target with a label there are basically two procedures:

1. By means of an argument or command option used to create the element to which the label will point. From this point of view, all the commands that create some kind of structure or text element open to being a reference target include an option called "`reference`" that is used to include the label. Occasionally, in place of the *option* the label is the content of the whole argument.

   We find a good example of what I am trying to say in the section commands that, as we know from (section 7.3), allow for several kinds of syntax. In the classic syntax the command is written as:

   ```
   \section[Label]{Title}
   ```

   and in the syntax specific to ConTEXt the command is written as

   ```
   \startsection
    [title=Title, reference=Label, ...   ]
   ```

   In both cases the command foresees the introduction of a label that will be associated with the section (or chapter, subsection, etc) in question.

   I said that this possibility is found in *all commands* that allow us to create a text element open to being a target of a reference. These are all text elements that can be numbered, including among others, sections, floating objects of all kinds (tables, images and similar), footnotes or end notes, quotations, numbered lists, descriptions, definitions, etc.

   > When the label is entered directly with an argument, and not as an option to which a value is assigned, it is possible with ConTEXt to associate several labels with a single target. For example:
   >
   > ```
   > \chapter[label1, label2, label3] {My chapter}
   > ```
   >
   > I am not clear what the advantage could be to have a number of different labels for the one target and suspect that it can be done not because it offers advantages but due to some *internal* requirement of ConTEXt applicable to certain kinds of arguments.

2. By means of the `\pagereference`, `\reference`, or `\textreference` commands whose syntax is:

   ```
   \pagereference[Label]
   \reference[Label]{Text}
   \textreference[Label]{Text}
   ```

   – The label created with `\pagereference` allows us to retrieve the page number.

   – Labels created with `\reference` and `\textreference` allow us to retrieve the page number as well as the text associated with them that is included as an argument.

In both \reference and \textreference the text that is linked to the label disappears as such from the final document at the point where the command is located (reference target), but can be retrieved and reappear at the point of origin of the reference.

I said earlier that each label is associated with certain information regarding the target point. What that information is depends on the type of label it is:

- All labels *remember* (in the sense that they make it possible to retrieve) the page number of the command that created them. For labels attached to sections that may have several pages, that number will be the page number where the section in question begins.
- Labels inserted with the command that creates a numbered text element (section, note, table, image, etc.) *remember* the number associated with that element (section number, note number, etc.)
- If this element has a *title*, as is the case, for example, for sections, but also tables if they have been inserted using the \placetable command, they will remember this title.
- Labels created with \pagereference only *remember* the page number.
- Those created with \reference or \textreference also remember the text associated with them that these commands take as an argument.

In fact I am not sure of the real difference between the \reference and \textreference commands. I think it is possible that the design of the three commands that allow the creation of labels attempts to run parallel with the three commands that allow the retrieval of information from the labels (which we will see in a moment); but the truth is that, according to my tests, \reference and \textreference seem to be redundant commands.

9.2.2 Commands at the reference point of origin for retrieving data from the target point

The commands that I will explain next retrieve information from the labels and, in addition, if our document is interactive, generate a connection to the reference target. But the important thing about these commands is the information that is retrieved from the label. If we only want to generate the connection, without retrieving any information from the label, we must use the \goto command explained in section 9.4.2.

A. Basic commands for retrieving information from a label

Bearing in mind that each label associated with a target point can store different items of information, it is logical that ConTEXt includes three different commands

for retrieving such information: depending on which information from a reference target point we want to retrieve, we use one or other of these commands:

- The `\at` command allows us to retrieve the label's page number.
- For labels that remember an element number (section number, note number, item number, table number, etc.) in addition to the page number, the `\in` command allows us to retrieve this number.
- Finally, for labels that remember a text associated with a label (a section title, image title inserted with `\placefigure`, etc.) the `\about` command allows us to retrieve this text.

The three `\at \in \about` commands have the same syntax:

```
\at{Text}[Label]
\in{Text}[Label]
\about{Text}[Label]
```

- Label is the label from which we want to retrieve information.
- Text is the text written just before the information we want to retrieve with the command. Between the text and the data of the label that the command retrieves, a non-separable space will be inserted and if the interactivity function is enabled in such a way that the command, besides retrieving the information, generates a link that allows us to jump to the target point, the text included as an argument will be part of the link (it will be clickable text).

So, in the following example we see how `\in` retrieves the section number and `\at` the page number.

```
In \in{section}[sec:target labels], that
begins on \at{page}
[sec:target labels], the
characteristics of labels used for
internal references are explained.
```

In section 9.2.1, that begins on page 167, the characteristics of labels used for internal references are explained.

Note that ConTEXt has automatically created hyperlinks (see section 9.3), and that the text taken as an argument by `\in` and `\at` is part of the link. But had we written it otherwise, the result would be:

```
In section \in{}[sec:target
labels], that begins on page \at{}
[sec:target labels], the
characteristics of labels used for
internal references are explained.
```

In section 9.2.1, that begins on page 167, the characteristics of labels used for internal references are explained.

The text remains the same, but the words *section* and *page* that precede the reference are not included in the link as they are no longer part of the command.

If ConTEXt is unable to find the label that the `\at`, `\in` or `\about` commands point to, no compiling error will result but where the information retrieved by these commands should appear in the final document we will see "??" written.

> There are two reasons why ConTEXt cannot find a label:
>
> 1. We made a mistake when writing it.
> 2. We are compiling only a part of the document, and the label points to the part not yet compiled (see sections 4.5.1 and 4.6).
>
> In the first case the error will need to be fixed. Therefore, it is a good idea when we finish compiling the complete document (and the second case is no longer possible), to look for all the appearances of "??" in the PDF to check that there are no *broken* references in the document.

B. Retrieving information associated with a label with the `\ref` command

Each of `\at`, `\in` and `\about` retrieve some elements of a label. Another command is available that allows us to rescue some element of the label that is indicated. This is the `\ref` command whose syntax is:

```
\ref[Element to retrieve][Label]
```

where the first argument can be:

- `text`: returns the text associated with a label.
- `title`: returns the title associated with a label.
- `number`: returns the number linked to a label. For example, in sections, the section number.
- `page`: returns the page number.
- `realpage`: returns the actual page number.
- `default`: returns what ConTEXt considers to be the *natural* element of the label. Generally this coincides with what is returned by `number`.

In fact, `\ref` is much more precise than `\at`, `\in` or `\about`, and thus, for example, it differentiates between the page number and the actual page number. The page number may not coincide with the actual number if, for example, the page numbering of the document started at 1500 (because this document is the continuation

of a previous one) or if the pages of the preamble were numbered with Roman numerals and seeing this the numbering was restarted. Similarly, `\ref` differentiates between the *text* and the *title* associated with a reference, something that `\about`, for example, does not do.

If `\ref` is used to get information from a label that lacks such information (e.g. the title of a label associated with a footnote), the command will return an empty string.

C. Detecting where the link leads to

ConTEXt also has two commands that are sensitive to *the link address*. With "link address" my intention is to determine whether the link target in the source file is found before or after the origin. For example: we are writing our document and we want to refer to a section that could still come before or after the one we are writing in the final table of contents. We just haven't decided yet. In this situation it would be useful to have a command that writes one or other depending on whether the target ultimately comes before or after the origin in the final document. For needs like this, ConTEXt provides the `\somewhere` command whose syntax is:

`\somewhere{Text if before}{Text if after}[Label]`.

For example, in the following text:

```
The hyperlink's address can also be detected by the \type{\somewhere} command.
This way we can also find chapters or other text elements
\somewhere {before}{after} [sec:references] and discuss their descriptions
in some other place \somewhere{before}{after} [sec:interactivity].
```

> The hyperlink's address can also be detected by the `\somewhere` command. This way we can find chapters or other text elements beforebeforeafteraftersec:references and discuss their descriptions in some other place beforebeforeafteraftersec:interactivity.
>
> For this example I have used two actual labels in this chapter in the source file.

Another command capable of detecting whether the label it points to comes before or after, is `\atpage` whose syntax is:

`\atpage[label]`

This command is quite similar to the previous one, but instead of allowing us to write the text ourselves, depending on whether the label comes before or after, `\atpage` inserts a default text for each of the two cases and, if the document is interactive, also inserts a hyperlink.

The text that `\atpage` inserts is the one associated with the "`precedingpage`" labels in case the *label* it takes as an argument is *before* the command, and "`hereafter`" in the opposite case.

When I arrived at this point, I was betrayed by a previous decision: in this chapter I decided to call what ConTEXt calls a "reference", a "label". It seemed clearer to me. But certain text fragments generated by ConTEXt commands, such as \atpage, are also called "labels" (this time in another sense). (See section 10.5.3). I hope the reader does not get confused. I think the context lets us properly distinguish which of the different meanings of *label* I am referring to in each case.

Therefore, we can change the text inserted by \atpage in the same way that we change the text of any other label:

```
\setuplabeltext[en][precedingpage=New text ]
\setuplabeltext[en][hereafter=New text ]
```

On this point I believe there is a small error in "ConTEXt Standalone" (the distribution I am using). Examining the names of the predefined labels in ConTEXt that can be changed with \setuplabeltext there are two pairs of labels that are candidates to be used by \atpage:

- "precedingpage" and "followingpage".
- "hencefore" and "hereafter".

We could assume that \atpage would use either the first or the second pair. But in fact, for items coming before, it uses "precedingpage" and for those following it uses "hereafter", which I think is inconsistent.

9.2.3 Automatic generation of prefixes to avoid duplicate labels

In a large document it is not always easy to avoid duplication of labels. It is therefore advisable to put some order into the way we choose which labels to use. One practice that helps is to use prefixes for the labels that will vary according to the type of label. For example "sec:" for sections, "fig:" for figures, "tbl:" for tables, etc.

With this in mind, ConTEXt includes a collection of tools that allow:

- ConTEXt itself to automatically generate labels for all the allowable elements.
- Every label generated manually to take a prefix, either one we have predetermined ourselves, or automatically generated by ConTEXt.

The detailed explanation of this mechanism is lengthy and, although they are undoubtedly useful tools, I do not think they are essential. Therefore, as they cannot be explained in a few words, I prefer not to explain them and refer to what is said about them in the ConTEXt reference manual or in the wiki on this matter.

9.3 Interactive electronic documents

Only electronic documents can be interactive; but not all electronic documents are. An *electronic* document is one that is stored in a computer file and can be

opened and read directly on screen. On the other hand, an electronic document that is equipped with utilities that allow the user *to interact* with it, is interactive; that is: we can do more than just read it. There is interactivity, for example, when the document has buttons that perform some action, or links through which we can jump to another point in the document, or to an external document; or when there are areas in the document where the user can write, or there are videos or audio clips that can be played, etc.

All documents generated by ConTEXt are electronic (since ConTEXt generates a PDF that is by definition an electronic document), but they are not always interactive. To provide them with interactivity it is necessary to expressly indicate this as shown in the next section.

Bear in mind, though, that although ConTEXt generates an interactive PDF, in order to appreciate this interactivity we need a PDF reader capable of it, since not all the PDF readers out there allow us to use hyperlinks, buttons and similar items proper to interactive documents.

9.3.1 Enabling interactivity in documents

ConTEXt does not use interactive functions by default unless expressly indicated, which is normally done in the preamble of the document. The command that enables this utility is:

```
\setupinteraction[state=start]
```

Normally this command would be used only once and in the document preamble when we want to generate an interactive document. But in fact we can use it as often as we want by altering the document's interactivity state. The "`state=start`" command enables interactivity, while "`state=stop`" disables it, so we can disable interactivity in some chapters or *parts* of our document where we want to do so.

> I can't think of any reason why we would want to have non-interactive parts in documents that are interactive. But what is important about the ConTEXt philosophy is that something be technically possible, even if we are unlikely to use it, so it offers a procedure for doing so. It is this philosophy that gives ConTEXt so many possibilities, and prevents a simple introduction like this from being *brief*.

Once interaction is established:

- Certain ConTEXt commands will already include hyperlinks. Thus:
  - The commands for creating tables of contents, which will be, in principle and unless expressly indicated otherwise, interactive, i.e. clicking on an

entry in the table of contents will jump to the page where the section in question begins.

  - The commands for internal references that we have seen in the first part of this chapter, where clicking on them automatically jumps to the reference target.

  - Footnotes and end notes where a click on the note anchor in the main body of the text will take us to the page where the note itself is written, and a click on the note mark in the note text will take us to the point in the main text where the call was made.

  - Etc.

- The possibility of using other commands specifically designed for interactive documents, such as presentations, is enabled. These employ numerous tools associated with interactivity such as buttons, menus, image overlays, embedded sound or video, etc. The explanation for all this would be too long and besides, presentations are a rather special kind of document. Therefore, in the following lines I will describe one feature associated with interactivity: hyperlinks.

9.3.2 Basic configuration for interactivity

`\setupinteraction`, in addition to enabling or disabling interaction, allows us to configure some matters related to it; mainly, but not only, the colour and style of the links. This is done through the following command options:

- `color`: controls the *normal* colour of links.

- `contrastcolor`: determines the colour of links where the target is on the same page as the origin. I recommend that this option always be set to the same content as the previous one.

- `style`: controls the link style.

- `title, subtitle, author, date, keyword`: The values assigned to these options will be converted into metadata of the PDF generated by ConTEXt.

- `click`: This option controls whether the link should be highlighted when it is clicked.

9.4 Hyperlinks to external documents

I will distinguish between commands that do not create the link but help to typeset the URL of the link, and commands that create the hyperlink. Let's look at them separately:

9.4.1 Commands that help typeset the hyperlinks but do not create them

URLs tend to be very long, and include characters of all types, even characters which are reserved characters in ConTEXt and cannot be used directly. In addition, when the URL must be displayed in the document, it is very difficult to typeset the paragraph, as the URL can exceed the length of a line and never includes blank spaces that can be used to insert a line break. In a URL, moreover, it is not reasonable to hyphenate words to insert line breaks, as the reader could hardly know whether or not the hyphenation actually forms part of the URL.

Therefore ConTEXt provides two utilities for *typesetting* URLs. The first is primarily for URLs that will be used internally, but will not actually be displayed in the document. The second is for URLs that have to be written in the text of the document. Let's look at them separately:

\useURL

This command allows us to write a URL in the preamble of the document, associating it with a name, so that when we want to use it in our document, we can invoke it by the name associated with it. It is especially useful with URLs that will be used several times throughout the document.

The command allows two usages:

1. `\useURL[Associated name][URL]`
2. `\useURL[Associated name] [URL] [] [Link text]`

- In the first version, the URL is simply associated with the name by which it will be invoked in our document. But then, to use the URL, we will have to indicate somehow, when invoking it, which clickable text will be shown in the document.

- In the second version the last argument includes the clickable text. The third argument exists in case we want to divide a URL into two parts, so that the first part contains the access address and the second part the name of the specific document or page that we want to open. For example: the address of the document that explains what ConTEXt is: http://www.pragma-ade.com/general/manuals/what-is-context.pdf. This address can be written in full in the second argument, leaving the third empty:

```
\useURL [WhatIsCTX]
  [http://www.pragma-ade.com/general/manuals/what-is-context.pdf]
  []
  [What is \ConTeXt?]
```

but we can also split it into two arguments:

```
\useURL [WhatIsCTX]
  [http://www.pragma-ade.com/general/manuals/]
  [what-is-context.pdf]
  [What is \ConTeXt?]
```

In both cases we will have associated that address with the word "`WhatIsCTX`", so that to include a link to that address, we use the command that we use to create the link; instead of the URL itself, we can simply write "`WhatIsCTX`".

If at any point in the text we want to reproduce a URL that we have associated with a name using `\useURL`, we can use the `\url[Associated name]` which inserts the URL associated with that name into the document. But this command, although it writes the URL, does not create any link.

> The format in which the URLs written using `\url` are displayed is not the one established in a general way by means of `\setupinteraction`, but the one specifically established for this command by means of `\setupurl`, which allows us to set the style (option `style`) and the colour (option `colour`).

`\hyphenatedurl`

This command is intended for URLs that will be written in the text of our document, and has ConTEXt include line breaks within the URL, if necessary, to correctly typeset the paragraph. Its format is:

```
\hyphenatedurl{URLaddress}
```

Despite the name of the `\hyphenatedurl` command, it does not hyphenate the name of the URL. What it does is to consider that certain characters common in URLs are good points to insert a line break before or after them. We can add the characters we want to the list of characters where a line break is allowed. We have three commands for this:

```
\sethyphenatedurlnormal{Characters}
\sethyphenatedurlbefore{Characters}
\sethyphenatedurlafter{Characters}
```

These commands add the characters they take as arguments to the list of characters that support line breaks before and after the list of characters that only support line breaks and those that only allow backward line breaks, respectively.

`\hyphenatedurl` can be used whenever a URL must be written that will appear in the final document as is. It can even be used as the last argument to `\useURL`

in the version of that command where the last argument picks up the clickable text to be displayed in the final document. For example:

```
\useURL [WhatIsCTX]
  [http://www.pragma-ade.com/general/manuals/what-is-context.pdf]
  []
  [\hyphenatedurl{http://www.pragma-ade.com/general/manuals/what-is-context.pdf}]
```

In the \hyphenatedurl argument all the reserved characters can be used except three which must be replaced by commands:

- % must be replaced by \letterpercent
- # must be replaced by \letterhash
- \ must be replaced by \letterescape or \letterbackslash.

Every time \hyphenatedurl inserts a line break it executes the \hyphenatedurlseparator command, which, by default, does nothing. But if we redefine it, a representative character is inserted in the URL in a similar way to what happens with normal words, where a hyphen is inserted to indicate that the word continues on the next line. For example:

```
\def\hyphenatedurlseparator{\curvearrowright}
```

will thus display the following particularly long web address:

https://support.microsoft.com/?scid=http://support.microsoft.com:80↷
/support/kb/articles/Q208/4/27.ASP&NoWebContent=1.

9.4.2 Commands that establish the link

To establish links to predefined URLs using \useURL we can use the command \from, which is limited to establishing the link, but does not write any clickable text. The default text in \useURL will be used as the link text. Its syntax is:

```
\from[Name]
```

where *Name* is the name previously associated with a URL using \useURL.

To create links and associate them with a clickable text that has not been previously defined, we have the \goto command which is used both to generate internal and external links. Its syntax is:

```
\goto{Clickable tex}[Target]
```

where *Clickable tex* is the text to be shown in the final document and where a mouse click will generate the jump, and *Target* can be:

- A label from our document. In this case \goto will generate the jump in a similar way as, for example, the \in or \at commands already examined. But

unlike those commands, no information associated with the label will be retrieved.

- The URL itself. In this case it must be expressly indicated that it is a URL by writing the command as follows:

  ```
  \goto{Clickable text}[url(URL)]
  ```

  where URL, in turn, can be the name previously associated with a URL by means of `\useURL`, or the URL itself, in which case, when writing the URL, we must ensure that the ConTEXt's reserved characters are written correctly in ConTEXt. Writing the URL according to ConTEXt rules will not affect the functionality of the link.

9.5 Creating bookmarks in the final PDF

PDF files can have an internal bookmark list of contents that allows the reader to see the contents of the document in a special window of the PDF viewer program, and to move through it by simply clicking on each of the sections and subsections.

By default, ConTEXt does not give the output PDF a bookmark list of contents, although getting it to do so is as simple as including the `\placebookmarks` command, whose syntax is:

```
\placebookmarks[List of sections]
```

where *List of sections* is a comma-separated list of the section levels that should appear in the list of contents.

Keep the following observations in mind regarding this command:

- According to my tests `\placebookmarks` does not work if it is in the preamble of the document. But, within the body of the document (between `\starttext` and `\stoptext`, or between `\startproduct` and `\stopproduct`), it doesn't matter where you place it: the bookmark list will also include the sections or subsections prior to the command. However, I believe that the most reasonable thing for a source file to be understood properly, is to place the command at the beginning.

- Section types defined by the user (with `\definehead`) are not always located in the right place in the bookmark list. It is preferable to exclude them.

- If the section title in any section includes an endnote or footnote, the text of the footnote shall be considered part of the bookmark.

- As an argument, instead of a list of sections, we can simply indicate the symbolic word "`all`" which, as its name indicates, will include all the sections;

however, according to my tests, this word, in addition to what are certainly sections, includes texts placed there with some non-sectioning commands, so the resulting list is somewhat unpredictable.

Not all PDF viewer programs allow us to view bookmarks; and many that do, do not have this feature activated by default. Therefore, to check the result of this function we must make sure that our PDF reader program supports this function and has it enabled. I think I remember that Acrobat, for example, does not show bookmarks by default, although there is a button on its toolbar to display them.

III
Particular issues

Chapter 10
Characters, words, text and horizontal space

The basic core element of all text documents is the character: characters are grouped into words, which in turn form lines that make up the paragraphs that make up pages.

The current chapter, starting with "*character*" explains some of ConTEXt's utilities relating to characters, words and text.

10.1 Getting characters not normally accessible from the keyboard

In a text file encoded as UTF-8 (see section 4.1) we can use any character or symbol, both of living languages and of many already extinct. But, as the possibilities of a keyboard are limited, most of the characters and symbols allowed in UTF-8 normally cannot be obtained directly from the keyboard. This is particularly the case with many diacritics, i.e. signs placed above (or below) certain letters, giving them a special value; but also with many other characters like maths symbols, traditional ligatures, etc. We can obtain many of these characters with ConTEXt by using commands.

10.1.1 Diacritics and special letters

Almost all Western languages have diacritics (with the important exception of English for the most part) and in general, keyboards can generate the diacritics corresponding to regional languages. Thus, a Spanish keyboard can generate all the diacritics needed for Spanish (basically accents and diaeresis) as well as some diacritics used in others languages such as Catalan (grave accents and cedillas) or French (cedillas, grave and circumflex accents); but not, for for example, some that are used in Portuguese, such as the tilde on some vowels in words like “navegação”.

TEX was designed in the United States where keyboards generally do not enable us to get diacritics; so Donald Knuth gave it a set of commands that enable us to obtain almost all the known diacritics (at least in languages using the Latin alphabet). If we use a Spanish keyboard, it does not make much sense to use these commands to obtain the diacritics that can be obtained directly from the keyboard. It is still important to know that these commands exist, and what they are, since Spanish (or Italian, or French...) keyboards do not let us generate all possible diacritics.

Name	Character	Abbreviation	Command
Acute accent	ú	`\'u`	`\uacute`
Grave accent	ù	`\`u`	`\ugrave`
Circumflex accent	û	`\^u`	`\ucircumflex`
Dieresis or umlaut	ü	`\"u`	`\udiaeresis, \uumlaut`
Tilde	ũ	`\~u`	`\utilde`
Macron	ū	`\=u`	`\umacron`
Breve	ŭ	`\u u`	`\ubreve`

Table 10.1 Accents and other diacritics

In table 10.1 we find the commands and abbreviations that allow us to obtain these diacritics. In all cases it is unimportant whether we use the command or the abbreviation. In the table, I have used the letter ‘u’ as an example, but these commands work with any vowel (most of them[1]) and also with some consonants and some semivowels.

- As most of the abbreviated commands are *control symbols* (see section 3.2), the letter on which the diacritic is to fall can be written immediately after the command, or separated from it. So, for example: to get the Portuguese ‘ã’ we

[1] Of the commands found in table 10.1 the tilde does not work with the letter ‘e’, and I don't know why.

can write the \=a or \=␣a characters.[1] But in the case of the breve (\u), when dealing with a *control word* the blank space is obligatory.

- In the case of the long version of the command, the letter on which the diacritic falls will be the first letter of the command name. So, for example \emacron will place a macron above a lower case 'e' (ē), \Emacron will do the same above an upper case 'E' (Ē), while \Amacron will do the same above an upper case 'A' (Ā).

While the commands in table 10.1 work with vowels and some consonants, there are other commands to generate some diacritics and special letters which only work on one or several letters. They are shown in table 10.2.

Name	Character	Abbreviation	Command
Scandinavian O	ø, Ø	\o, \O	
Scandinavian A	å, Å	\aa, \AA, {\r a}, {\r A}	\aring, \Aring
Polish L	ł,	\l, \L	
German Eszett	ß	\ss, \SS	
'i' and 'j' wihtout a point	ı, ȷ	\i, \j	
Hungarian Umlaut	ű, Ű	\H u, \H U	
Cedilla	ç, Ç	\c c, \c C	\ccedilla, \Ccedilla

Table 10.2 More diacritics and special letters

I would like to point out that some of the commands in the above table generate the characters from other characters, while other commands only work if the font we are using has expressly provided for the character in question. So where German Eszett (ß) is concerned, the table shows two commands but only one character, because the font I am using here for this text only provides for the upper case version of German Eszett (something quite common).

That's probably why I can't get the Scandinavian A in upper case either although "{\r A}" and \Aring work correctly.

The Hungarian umlaut also works with the letter 'o', and the cedilla with the letters 'k', 'l', 'n', 'r', 's' and 't', in lower or upper case, respectively. The commands to be used are \kcedilla, \lcedilla, \ncedilla ... respectively.

10.1.2 Traditional ligatures

A ligature is formed by the union of two or more graphemes that are usually written separately. This "fusion" between two characters often started out as a kind of shorthand in handwritten texts, until finally they achieved a certain typographic independence. Some of them were even included among the characters that are

[1] Remember that in this document we are representing blank spaces, when it is important that we see them, with the '␣'.

usually defined in a typographic font, such as the ampersand, '&', which began as a contraction of the Latin copula (conjunction) "et", or the German Eszett (ß), which, as its name indicates, began as a combination of an 's' and 'z'. In some font designs, even today, we can trace the origins of these two characters; or maybe I see them because I know they're there. In particular, with the Pagella font for '&' and with Bookman for 'ß'.

As an exercise I suggest (after reading Chapter 6, where it explains how to do it) try representing these characters with these fonts at a size large enough (for example, 30 pt) to be able to work out their components.

Other traditional ligatures which did not become so popular, but are still used occasionally today, are the Latin endings "oe" and "ae" which were occasionally written as 'œ' or 'ae' to indicate that they formed a diphthong in Latin. These ligatures can be achieved in ConTEXt with the commands found in table 10.3

Ligature	Abbreviation	Command
æ, Æ	`\ae`, `\AE`	`\aeligature`, `\AEligature`
œ, Œ	`\oe`, `\OE`	`\oeligature`, `\OEligature`

Table 10.3 Traditional ligatures

A ligature that used to be traditional in Spanish (Castilian) and that is not usually found in fonts today, is 'Ð': a contraction involving 'D' and 'E'. As far as I know there is no command in ConTEXt that lets us use this,[1] but we can create one, as explained in section 10.1.5.

Along with the previous ligatures, which I have called *traditional* because they come from handwriting, after the invention of the printing press certain printed text ligatures developed which I will call "typographical ligatures" considered by ConTEXt to be font utilities and which are managed automatically by the program, although we can influence how these font utilities are handled (including ligatures) with `\definefontfeature` (not explained in this introduction).

10.1.3 Greek letters

It is common to use Greek characters in mathematical and physics formulas. This is why ConTEXt included the possibility of generating all of the Greek alphabet, upper and lower case. Here the command is built on the English name for the Greek letter in question. If the first character is written in lower case we will have the lower case Greek letter and if it is written in capital letters we will get the Greek letter in upper case. For example, the command `\mu` will generate the lower case version of this letter (μ) while `\Mu` will generate the upper case version (M).

[1] In LaTeX, by contrast, we can use the `\DH` command implemented by the "`fontenc`" package.

In table 10.4 we can see which command generates each of the letters in the Greek alphabet, lower case and upper case.

English name	Character (lc/uc)	Commands (lc/uc)
Alpha	α, Α	`\alpha, \Alpha`
Beta	β, Β	`\beta, \Beta`
Gamma	γ, Γ	`\gamma, \Gamma`
Delta	δ, Δ	`\delta, \Delta`
Epsilon	ϵ, ε, Ε	`\epsilon, \varepsilon, \Epsilon`
Zeta	ζ, Ζ	`\zeta, \Zeta`
Eta	η, Η	`\eta, \Eta`
Theta	θ, ϑ, Θ	`\theta, \vartheta, \Theta`
Iota	ι, Ι	`\iota, \Iota`
Kappa	κ, ϰ, Κ	`\kappa, \varkappa, \Kappa`
Lambda	λ, Λ	`\lambda, \Lambda`
Mu	μ, Μ	`\mu, \Mu`
Nu	ν, Ν	`\nu, \Nu`
Xi	ξ, Ξ	`\xi, \Xi`
Omicron	ο, Ο	`\omicron, \Omicron`
Pi	π, ϖ, Π	`\pi, \varpi, \Pi`
Rho	ρ, ϱ, Ρ	`\rho, \varrho, \Rho`
Sigma	σ, ς, Σ	`\sigma, \varsigma, \Sigma`
Tau	τ, Τ	`\tau, \Tau`
Ypsilon	υ, Υ	`\upsilon, \Upsilon`
Phi	ϕ, φ, Φ	`\phi, \varphi, \Phi`
Chi	χ, Χ	`\chi, \Chi`
Psi	ψ, Ψ	`\psi, \Psi`
Omega	ω, Ω	`\omega, \Omega`

Table 10.4 Greek alphabet

Note how for lower case versions of some characters (epsilon, kappa, theta, pi, rho, sigma and phi) there are two possible variants.

10.1.4 Various symbols

Together with the characters we have just seen, TEX (and therefore ConTEXt as well) offers commands for generating any number of symbols. There are many such commands. I have provided an extended although incomplete list in Appendix B.

10.1.5 Defining characters

If we need to use any characters not accessible from our keyboard, we can always find a web page with these characters and copy them into our source file. If we are using UTF-8 encoding (as recommended) this will almost always work. But also in the ConTEXt wiki there is a page with heaps of symbols that can be simply copied and pasted into our document. To get them, click on this link.

However, if we need to use one of the characters in question more than once, then copy-paste is not the most efficient way to do so. It would be preferable to define

the character so that it is associated with a command that will generate it each time. To do this we use \definecharacter whose syntax is:

\definecharacter *Name Character*

where

- **Name** is the name associated with the new character. It should not be the name of an existing command, as this would overwrite that command.
- **Character** is the character generated each time we run *Name*. There are three ways we can indicate this character:
  - By simply writing it or pasting it into our source file (if we have copied it from another electronic document or web page).
  - By indicating the number associated with that character in the font we are currently using. In order to see the characters included in the font, and the numbers associated with them, we can use the \showfont[*Font name*] command.
  - Building the new character with one of the composite character building commands that we will see immediately following.

As an example of the first usage, let's return for the moment to the sections dealing with ligatures (10.1.2). There I spoke about a traditional ligature in Spanish that we can't usually find in fonts today: 'Ð'. We could associate this character, for example, with the \decontract command so that the character will be generated whenever we write \decontract. We do this with:

```
\definecharacter decontract Ð
```

> To build a new character that is not in our font, and cannot be obtained from the keyboard, as is the case of the example I have just given, first we must find some text where that character is found, copy it and be able to paste it into our definition. In the actual example I have just given, I originally copied the 'Ð' from Wikipedia.

ConTEXt also includes some commands that allow us to create composite characters and that can be used in combination with \definecharacter. By composite characters I mean characters that also have diacritics. The commands are as follows:

```
\buildmathaccent Accent Character
\buildtextaccent Accent Character
\buildtextbottomcomma Character
\buildtextbottomdot Character
\buildtextcedilla Character
\buildtextgrave Character
\buildtextmacron Character
```

\buildtextognek Character

For example: as we already know, by default ConTEXt only has commands for writing certain letters with a cedilla (c, k, l, n, r, s y t) that are usually incorporated into fonts. If we wanted to use a 'b' we could use the \buildtextcedilla command as follows:

```
\definecharacter bcedilla {\buildtextcedilla b}
```

This command will create the new \bcedilla command that will generate a 'b' with a cedilla: 'ḇ'. These commands literally "build" the new character that will be generated even though our font doesn't have it. What these commands do is to superimpose one character over another then give a name to that superimposition.

> In my tests I was unable to make \buildmathaccent or \buildtextognek work. So I will no longer mention them from here on.

\buildtextaccent takes two characters as arguments and superimposes one on the other, raising one of them slightly. Although it is called "buildtextaccent", it is not essential that any of the characters taken as arguments is an accent; but the overlap will give better results if it is, because in this case, by superimposing the accent on the character the accent is less likely to overwrite the character. On the other hand, the overlapping of two characters that have the same baseline under normal conditions is affected by the fact that the command slightly raises one of the characters above the other. This is why we cannot use this command, for example, to get the contraction 'Ð' mentioned above, because if we write

```
\definecharacter decontract {\buildtextaccent D E}
```

in our source file, the slight elevation above the 'D' baseline that this command produces means that the ("Ɖ") effect it produces is not very good. But if the height of the characters allows it we could create a combination. For example,

```
\definecharacter unusual {\buildtextaccent \_ "}
```

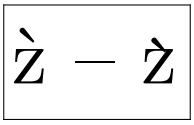
would define the '"_' character that would be associated with the \unusual command.

The rest of the build commands takes a single argument – the character that the diacritic generated by each command will be added to. Below I will show an example of each of them, built on the letter 'z':

- \buildtextbottomcomma adds a comma beneath the character it takes as an argument ('ẓ').
- \buildtextbottomdot adds a point beneath the character it takes as an argument ('ż').

- \buildtextcedilla adds a cedilla beneath the character it takes as an argument (‘z̧’).
- \buildtextgrave adds a grave accent above the character it takes as an argument (‘z̀’).
- \buildtextmacron adds a small bar beneath the character it takes as an argument (‘z̲’).

At first sight, \buildtextgrave seems redundant given that we have \buildtextaccent; However, if you check the grave accent generated with the first of these two commands, it looks a little better. The following example shows the result of both commands, at a sufficient font size to appreciate the difference:



10.1.6 Use of predefined symbol sets

“ConTEXt Standalone” includes, along with ConTEXt itself, a number of predefined symbol sets we can use in our documents. These sets are called “cc”, “cow”, “fontawesome”, “jmn”, “mvs” and “nav”. Each of these sets also includes some subsets:

- **cc** includes “cc”.
- **cow** includes “cownormal” and “cowcontour”.
- **fontawesome** includes “fontawesome”.
- **jmn** includes “navigation 1”, “navigation 2”, “navigation 3” and “navigation 4”.
- **mvs** includes “astronomic”, “zodiac”, “europe”, “martinvogel 1”, “martinvogel 2” and “martinvogel 3”.
- **nav** includes “navigation 1”, “navigation 2” and “navigation 3”.

> The wiki also mentions a set called **was** that includes “wasy general”, “wasy music”, “wasy astronomy”, “wasy astrology”, “wasy geometry”, “wasy physics” and “wasy apl”. But I couldn't find them in my distribution, and my tests to attempt to get at them failed.

To see the specific symbols contained in each of these sets, the following syntax is used:

```
\usesymbols[Set]
\showsymbolset[Subset]
```

For example: if we want to see the symbols included in “mvs/zodiac”, then in the source file we need to write:

```
\usesymbols[mvs]
\showsymbolset[zodiac]
```

and we will get the following result:

Aquarius	♒	♒
Aries	♈	♈
Cancer	♋	♋
Capricorn	♑	♑
Gemini	♊	♊
Leo	♌	♌
Libra	♎	♎
Pisces	♓	♓
Sagittarius	♐	♐
Scorpio	♏	♏
Taurus	♉	♉
Virgo	♍	♍

Note that the name of each symbol is indicated as well as the symbol. The `\symbol` command allows us to use any of the symbols. Its syntax is:

`\symbol[Subset][SymbolName]`

where subset is one of the subsets associated with any of the sets we have previously loaded with `\usesymbols`. For example, if we wanted to use the astrological symbol associated with Aquarius (found in mvs/zodiac) we would need to write

```
\usesymbols[mvs]
\symbol[zodiac][Aquarius]
```

which will give us the "♒", and this, for all intents and purposes, will be treated as a "character" and is therefore affected by the font size that is active when printed. We can also use `\definecharacter` to associate the symbol in question with a command. For example

```
\definecharacter Aries {\symbol[zodiac][Aries]}
```

will create a new command called `\Aries` that will generate the character "♈".

We could also use these symbols, for example, in an itemize environment. For example:

```
\usesymbols[mvs]
\definesymbol[1][{\symbol[martinvogel 2][PointingHand]}]
\definesymbol[2][{\symbol[martinvogel 2][CheckedBox]}]
\startitemize[packed]
\item item \item item
 \startitemize[packed]
```

```
 \item item \item item
 \stopitemize
\item item
\stopitemize
```

will produce

- ☞ item
- ☞ item
  - ☑ item
  - ☑ item
- ☞ item

10.2 Special character formats

Strictly speaking, it is *format* commands that affect the font used, its size, style or variant. These commands are explained in Chapter 6. However, seen more *broadly*, we can also consider the commands that somehow change the characters they take as an argument (thus altering their appearance) to be format commands. We will look at some of these commands in this section. Others, such as underlined or lined text with lines above or below the text (e.g. where we want to provide space to answer a question) will be seen in section 12.5.

10.2.1 Upper case, lower case and fake small caps

Letters themselves can be upper case or lower case. For ConTEXt, upper case and lower case letters are different characters, so in principle it will typeset the letters just as it finds them written. However, there is a group of commands which allow us to ensure that the text they take as an argument is always written in upper or lower case:

- `\word{text}`: converts the text taken as an argument into lower case.
- `\Word{text}`: converts the first letter of the text taken as an argument into upper case.
- `\Words{text}`: converts the first letter of each of the words taken as an argument into upper case; the rest are in lower case.
- `\WORD{text}` or `\WORDS{text}`: writes the text taken as an argument in upper case.

Very similar to these commands are `\cap` and `\Cap`: they also capitalise the text they take as an argument, but then apply a scaling factor to it equal to that applied by the 'x' suffix in font change commands (see section 6.4.2) so that, in most fonts, the caps will be the same height as lower case letters, thus giving us a

kind of *fake small caps* effect. Compared to genuine small caps (see section 6.5.2) these have the following advantages:

1. `\cap` and `\Cap` will work with any font, by contrast with genuine small caps that only work with fonts and styles that expressly include them.

2. True small caps, on the other hand, are a variant of the font which, as such, is incompatible with any other variant such as bold, italic, or slanted. However, `\cap` and `\Cap` are fully compatible with any font variant.

The difference between `\cap` and `\Cap` is that while the former applies the scaling factor to all the letters of the words that make up its argument, `\Cap` does not apply any scaling to the first letter of each word, thus achieving an effect similar to what we get if we use real capitals in a text in small caps. If the text taken as an argument in 'caps' consists of several words, the size of the capital letter in the first letter of each word will be maintained.

Thus, in the following example

```
The UN, whose \Cap{president} has his
office at \cap{uN} headquarters...
```

The UN, whose PRESIDENT has his office at UN headquarters...

we need to note, first of all, the difference in size between the first time we write "UN" (in upper case) and the second time (in small caps, "UN"). In the example, I wrote `\cap{uN}` the second time so we can see that it does not matter if we write the argument that `\cap` takes in upper or lower case: the command converts all letters into upper case and then applies a scaling factor; by contrast with `\Cap` that does not scale the first letter.

These commands can also be *nested*, in which case the scaling factor would be applied once more, resulting in a further reduction, as in the following example where the word "capital" in the first line is scaled yet again:

```
\cap{People who have amassed their
\cap{capital} at the expense of others
are more often than not
{\bf decapitated} in revolutionary
times}.
```

PEOPLE WHO HAVE AMASSED THEIR CAPITAL AT THE EXPENSE OF OTHERS ARE MORE OFTEN THAN NOT **DECAPITATED** IN REVOLUTIONARY TIMES.

The `\nocap` command applied to a text to which `\cap` is applied, cancels out the `\cap` effect in the text that is its argument. For example:

```
\cap{When I was One I had just begun,
when I was Two I was \nocap{nearly}
new (A.A. Milne)}.
```

WHEN I WAS ONE I HAD JUST BEGUN, WHEN I WAS TWO I WAS nearly NEW (A.A. MILNE).

We can configure how \cap works with \setupcapitals and we can also define different versions of the command, each with its own name and specific configuration. This we can do with \definecapitals.

Both commands work in a similar way:

```
\definecapitals[Name][Configuration]
\setupcapitals[Name][Configuration]
```

The "Name" parameter in \setupcapitals is optional. If it is not used, the configuration will affect the \cap command itself. If it is used, we need to give the name we previously assigned in \definecapitals to some actual configuration.

In either of the two commands the configuration allows for three options: "title", "sc" and "style" the first and second allowing "yes" and "no" as values. With "title" we indicate whether the capitalisation will also affect titles (which it does by default) and with "sc" we indicate whether the command should be genuine small caps ("yes"), or fake small caps ("no"). By default it uses fake small caps which has the advantage that the command works even if you are using a font that has not implemented small caps. The third value "style" allows us to indicate a style command to be applied to the text affected by the \cap command.

10.2.2 Superscript or subscript text

We already know (see section 3.1) that in maths mode, the reserved characters "_" and "^" will convert the character or group that immediately follows into a superscript or subscript. To achieve this effect outside of maths mode, ConTEXt includes the following commands:

- \high{Text}: writes the text it takes as an argument as a superscript.
- \low{Text}: writes the text it takes as an argument as a subscript.
- \lohi{Subscript}{Superscript}: writes both arguments, one above the other: on the bottom the first argument, and on top the second, which brings about a curious effect:

```
\lohi{below}{above}
```

above
below

10.2.3 Verbatim text

The Latin expression *verbatim* (from *verbum* = *word* + the suffix *atim*), which could be translated as "literally" or "word for word", is used in text processing programs like ConTEXt to refer to fragments of text that should not be processed at all, but should be dumped, as written, into the final file. ConTEXt uses the command `\type` for this, intended for short texts that do not occupy more than one line and the `typing` environment intended for texts of more than one line. These commands are widely used in computer books to show code fragments, and ConTEXt formats these texts in monospaced letters like a typewriter or a computer terminal would. In both cases the text is sent to the final document without *processing*, which means that they can use reserved characters or special characters that will be transcribed *as is* in the final file. Likewise, if the argument of `\type`, or the content of `\starttyping` includes a command, this will be *written* in the final document, but not executed.

The `\type` command has, besides, the following peculiarity: its argument *can* be contained within curly brackets (as is normal in ConTEXt), but any other character can be used to delimit (surround) the argument.

When ConTEXt reads the `\type` command it assumes that the character which is not a blank space immediately following the name of the command will act as a delimiter of its argument; so it considers that the contents of the argument begin with the next character, and end with the character before the next appearance of the *delimiter*.

Some examples will help us to understand this better:

```
\type 1Tweedledum and Tweedledee1
\type |Tweedledum and Tweedledee|
\type zTweedledum and Tweedledeez
\type (Tweedledum and Tweedledee(
```

Note that in the first example, the first character after the command name is a '1', in the second a '|' and in the third a 'z'; so: in each of these cases ConTEXt will consider that the argument of `\type` is everything between that character and the next appearance of the same character. The same is true for the last example, which is also very instructive, because in principle we could assume that if the opening delimiter of the argument is a '(', the closing one should be a ')', but it is not, because '(' and ')' are different characters and `\type`, as I said, searches for a closing character delimiter which is the same as the character used at the start of the argument.

There are only two cases where `\type` allows the opening and closing delimiters to be different characters:

- If the opening delimiter is the '{' character, it thinks the closing delimiter will be '}'.
- If the opening delimiter is '<<', it thinks that the closing delimiter will be '>>'. This case is also unique in that two consecutive characters are being used as delimiters.

However: the fact that `\type` allows any delimiter does not mean that we should use "weird" delimiters. From the point of view of the *readability* and *comprehensibility* of the file source,

it is best to delimit the argument of \type with curly brackets where possible, as is normal with ConTEXt; and when this is not possible, because there are curly brackets in the \type argument, use a symbol: preferably one that is not a ConTEXt reserved character. For example: \type *This is a closing curly bracket: '}'*.

Both \type and \starttyping can be configured with \setuptype and \setuptyping. We can also create a customised version of these with \definetype and \definetyping. Regarding the actual configuration options for these commands, I refer to "setup-en.pdf" (in the directory tex/texmf-context/doc/context/documents/general/qrcs.

Two very similar commands to \type are:

- \typ: works similarly to \type, but does not disable hyphenation.
- \tex: a command intended for writing texts about TEX or ConTEXt: it adds a backspace before the text it takes as an argument. Otherwise, this command differs from \type in that is processes some of the reserved characters it finds in the text it takes as an argument. In particular, curly brackets inside \tex will be treated in the same way they are usually treated in ConTEXt.

10.3 Character and word spacing

10.3.1 Automatically setting horizontal space

The space between different characters and words (called *horizontal space* in TEX) is normally set automatically by ConTEXt:

- The space between the characters that make up a word is defined by the font itself, which, except in fixed-width fonts, usually uses a greater or lesser amount of white space depending on the characters to be separated, and so, for example, the space between an 'A' and a 'V' ('AV') is usually less than the space between an 'A' and an 'X' ('AX'). However, apart from these possible variations that depend on the combination of letters concerned and predefined by the font, the space between the characters that make up a word is, in general, a fixed and invariable measure.
- By contrast, the space between words on the same line can be more elastic.
  - In the case of words in a line whose width must be the same as that of the rest of the lines in the paragraph, the variation of the spacing between words is one of the mechanisms that ConTEXt uses to obtain lines of equal width, as explained in more detail in section 11.3. In these cases, ConTEXt will establish exactly the same horizontal space between all the words in

the line (except for the rules below), while ensuring that the space between words in the different lines of the paragraph is as similar as possible.

– However, in addition to the need to stretch or shrink the spacing between words in order to justify the lines, depending on the active language, ConTEXt takes certain typographical rules into consideration whereby in certain places the typographical tradition associated with that language adds some extra white space, as is the case, for example, in some parts of the English typographical tradition, which adds extra white space after a full stop.

> These extra white spaces work for English and possibly for some other languages (though it is also true that in many instances, publishers in English nowadays choose not to have extra space after a full stop) but not for Spanish where the typographical tradition is different. So we can temporarily enable this function with `\setupspacing[broad]` and disable it with `\setupspacing[packed]`. We could also change the default configuration for Spanish (and for that matter for any other language including English), as explained in section 10.5.2.

10.3.2 Altering the space between characters within a word

Altering the default space for the characters that make up a word is considered very bad practice from a typographical point of view, except in titles and headings. However, ConTEXt provides a command to alter this space between the characters in a word:[1] `\stretched`, whose syntax is as follows:

```
\stretched[Configuration]{Text}
```

where *Configuration* allows any of the following options:

- `factor`: an integer or decimal number representative of the spacing to be obtained. It should not be too high a number. A factor of 0.05 is already visible to the naked eye.
- `width`: indicates the total width that the text submitted to the command must have, in such a way that the command itself will calculate the necessary spacing to distribute the characters in that space.

> According to my tests, when the width established with the `width` option is less than that required to represent the text with a *factor* equal to 0.25, the *width* option and this factor are ignored. I guess that's because `\stretched` allows us only *to increase* the space between the characters in a word, not reduce it. But I don't understand why the width

[1] It is very typical of the philosophy of ConTEXt to include a command to do something that the ConTEXt documentation itself advises against doing. Although typographical perfection is sought, the aim is also to give the author absolute control over the appearance of his or her document: whether it is better or worse is, in short, his or her responsibility.

required to represent the text with a factor of 0.25 is used as a minimum measure for the `width` option, and not the *natural width* of the text (with a factor of 0).

- **`style`**: style command or commands to apply to the text taken as an argument.
- **`color`**: the colour in which the text taken as an argument will be written.

So in the following example we can see graphically how the command would work when applied to the same sentence, but with different widths:

```
\stretched[width=4cm]{\bf test text}
\stretched[width=6cm]{\bf test text}
\stretched[width=8cm]{\bf test text}
\stretched[width=9cm]{\bf test text}
```

t e s t t e x t
t e s t t e x t
t e s t t e x t
t e s t t e x t

In this example it can be seen that the distribution of the horizontal space between the different characters is not uniform. The 'x' and 't' in "text" and the 'e' and 'b' in "test", always appear much closer together than the other characters. I haven't been able to find out why this happens.

Applied without arguments, the command will use the full width of the line. On the other hand, within the text that is the argument to this command, the command `\\` is redefined and instead of a line break, it inserts horizontal space. For example:

```
\stretched{test\\text}
```

t e s t t e x t

We can customise the default configuration of the command with `\setupstretched`.

There is no `\definestretched` command that would allow us to set customised configurations associated with a command name, however, in the official list of commands (see section 3.6) it says that `\setupstretched` comes from `\setupcharacterkerning` and there is a `\definecharacterkerning` command. In my tests, however, I have not managed to set any customised configuration for `\stretched` by means of the latter, although I must admit that I have not spent much time trying to do so either.

10.3.3 Commands for adding horizontal space between words

We already know that to increase the space between words it is of no use to add two or more consecutive blank spaces, since ConTEXt absorbs all consecutive blank spaces, as explained in section 4.2.1. If we wish to increase the space between words, we need to go to one of the commands that allows us to do this:

- `\,` inserts a very small blank space (called a thin space) in the document. It is used, for example, to separate thousands in a set of numbers (e.g. 1,000,000),

or to separate a single inverted comma from double inverted commas. For example: "`1\,473\,451`" will produce "1 473 451".

- `\space` or "`\␣`" (a backslash followed by a blank space which, since it is an invisible character, I have represented as "␣") introduces an additional blank space.
- `\enskip`, `\quad` and `\qquad` insert a blank space in the document of half an *em*, 1 *em* or 2 *ems* respectively. Remember that the *em* is a measure dependent on the size of the font and is equivalent to the width of an 'm', which normally coincides with the size in points of the font. So, using a 12 point font, `\enskip` gives us a space of 6 points, `\quad` gives us 12 points and `\qquad` gives us 24 points.

Along with these commands which give us blank space in precise measurements, the `\hskip` and `\hfill` commands introduce horizontal space of varying dimensions:

`\hskip` allows us to indicate exactly how much blank space we want to add. Thus:

```
This is \hskip 1cm 1 centimetre\\
This is \hskip 2cm 2 centimetres\\
This is \hskip 2.5cm 2.5 centimetres\\
```

This is 1 centimetre
This is 2 centimetres
This is 2.5 centimetres

The space indicated may be negative, which will cause one text to be superimposed over another. Thus:

```
This is farce rather than
\hskip -1cm comedy
```

This is farce rathercomedy than

`\hfill`, for its part, introduces as much white space as necessary to occupy the entire line, allowing us to create interesting effects such as right-aligned text, centred text or text on both sides of the line as shown in the following example:

```
\hfill On the right\\
On both\hfill sides
```

On the right
On both sides

10.4 Compound words

By "compound words" in this section I mean words that are formally understood to be one word, rather than words that are simply conjoined. It is not always an

easy distinction to understand: "rainbow" is clearly made up of two words ("rain + bow") but no English speaker would think of the combined terms in any other way than as a single word. On the other hand, we have words that are sometimes combined with the help of a hyphen or backslash. The two words have distinct meanings and uses but are conjoined (and may in some cases become a single word, but not yet!). So, for example, we can find words like "French–Canadian" or "(inter)communication" (though we may well also find "intercommunication" and discover that the speaking public has finally accepted the two words to be a single word. That is how language evolves).

Compound words present ConTEXt with some problems mainly connected with their potential hyphenation at the end of a line. If the joining element is a hyphen, then from a typographical perspective there is no hyphenation problem at the end of a line at that point, but we would need to avoid a second hyphenation in the second part of the word since that would leave us with two consecutive hyphens which could cause comprehension difficulties.

The "`||`"command is available to tell ConTEXt that two words make up a compound word. This command, exceptionally, does not begin with a backslash, and allows two different usages:

- We can use two consecutive vertical bars (pipes) and write, for example, "`Spanish||Argentine`".
- The two vertical bars can have the joining /separating item between two words enclosed between them, as in, for example, "`joining|/|separating`".

In both cases, ConTEXt will know that it is dealing with a compound word, and will apply the appropriate hyphenation rules for this type of word. The difference between using the two consecutive vertical bars (pipes), or framing the word separator with them, is that in the first case, ConTEXt will use the separator that is predefined as `\setuphyphenmark`, or in other words the hyphen, which is the default ("`--`"). So if we write "`picture||frame`", ConTEXt will generate "Picture–frame".

With `\setuphyphenmark` we can change the default separator (in the case where we need two pipes). The values allowed for this command are "`--, ---, -, , (, ), =, /`". Bear in mind, however, that the "`=`" value becomes an em dash (the same as "`---`").

The normal use of "`||`" is with hyphens, since this is what is normally used between composite words. But occasionally the separator could be a parenthesis, if, for example, we want "(inter)space", or it could be a forward slash, as in "input/output". In these cases, if we want the normal hyphenation rules for composite words to apply, we could write "`(inter|)|space`" or "`input|/|output`". As I said earlier,

"|=|" is considered to be an abbreviation of "|---|" and inserts an em dash as a separator (—).

10.5 The language of the text

Characters form words which normally belong to some language. It is important for ConTEXt to know the language we are writing in, because a number of important things depend on this. Mainly:

- Word hyphenation.
- The output format of certain words.
- Certain typesetting matters associated with the typesetting tradition of the language in question.

10.5.1 Setting and changing the language

ConTEXt assumes that the language will be English. Two procedures can change this:

- By using the \mainlanguage command, used in the preamble to change the main language of the document.

- By using the \language command, aimed at changing the active language at any point in the document.

Both commands expect an argument consisting of any language identifier (or code). To identify the language, we use either the two-letter international language code set out in ISO 639-1, which is the same as that used, for example, on the web, or the English name of the language in question, or sometimes some abbreviation of the name in English.

In table 10.5 we find a complete list of languages supported by ConTEXt, along with the ISO code for each of the languages in question as well as, where appropriate, the code for certain language variants expressly provided for.[1]

[1] Table 10.5 has a summary of the list obtained with the following commands:
\usemodule[languages-system]
\loadinstalledlanguages
\showinstalledlanguages
Should you be reading this document long after it was written (2020) it is possible that ConTEXt will have incorporated additional languages, so it would be a good idea to use these commands to show an updated list of languages

Language	**ISO code**	**Language** (variants)
Afrikaans	af, afrikaans	
Arabic	ar, arabic	ar-ae, ar-bh, ar-dz, ar-eg, ar-in, ar-ir, ar-jo, ar-kw, ar-lb, ar-ly, ar-ma, ar-om, ar-qa, ar-sa, ar-sd, ar-sy, ar-tn, ar-ye
Catalan	ca, catalan	
Czech	cs, cz, czech	
Croatian	hr, croatian	
Danish	da, danish	
Dutch	nl, nld, dutch	
English	en, eng, english	en-gb, uk, ukenglish, en-us, usenglish
Estonian	et, estonian	
Finnish	fi, finnish	
French	fr, fra, french	
German	de, deu, german	de-at, de-ch, de-de
Greek	gr, greek	
Greek (ancient)	agr, ancientgreek	
Hebrew	he, hebrew	
Hungarian	hu, hungarian	
Italian	it, italian	
Japanese	ja, japanese	
Korean	kr, korean	
Latin	la, latin	
Lithuanian	lt, lithuanian	
Malayalam	ml, malayalam	
Norwegian	nb, bokmal, no, norwegian	nn, nynorsk
Persian	pe, fa, persian	
Polish	pl, polish	
Portuguese	pt, portughese	pt-br
Romanian	ro, romanian	
Russian	ru, russian	
Slovak	sk, slovak	
Slovenian	sl, slovene, slovenian	
Spanish	es, sp, spanish	es-es, es-la
Swedish	sv, swedish	
Thai	th, thai	
Turkish	tr, turkish	tk, turkmen
Ukranian	ua, ukrainian	
Vietnamese	vi, vietnamese	

Table 10.5 Language support in ConTEXt

So, for example, to set Spanish (Castilian) as the main language of the document we could use any of the three that follow:

```
\mainlanguage[es]
\mainlanguage[spanish]
\mainlanguage[sp]
```

To enable a particular language *inside* the document, we can use either the `\language[Language code]` command, or a specific command to activate that language. So, for example, `\en` activates English, `\fr` activates French, `\es` Spanish, or `\ca` Catalan. Once an actual language has been activated, it remains so until we expressly switch to another language, or the group in which the language was activated is then closed. Languages work, therefore, just like font change commands. Note, however, that the language set by the `\language` command or by one of its

abbreviations (\en, \fr, \de, etc.) does not affect the language in which labels are printed (see section 10.5.3).

> Although it may be laborious to mark the language of all the words and expressions we use in our document that do not belong to the main language of the document, it is important to do so if we want to obtain a properly typeset final document, especially in professional work. We should not mark all the text, but only the part that does not belong to the main language. Sometimes it is possible to automate the marking of the language by using a macro. For example, for this document in which ConTEXt commands are continuously being quoted, the original language of which is English, I have designed a macro which, in addition to writing the command in the appropriate format and colour, marks it as an English word. In my professional work, where I need to quote a lot of French and Italian bibliography, I have incorporated a field in my bibliographic database to pick up the language of the work, so that I can automate the language indication in the quotations and lists of bibliographical references.
>
> If we are using two languages that use different alphabets in the same document (for example, English and Greek, or English and Russian), there is a trick that will prevent us from having to mark the language of expressions built with the alternative alphabet: modify the main language setting (see next section) so that it also loads the default hyphenation patterns for the language that uses a different alphabet. For example, if we want to use English and ancient Greek, the following command would save us from having to mark language of the texts in Greek:
>
> ```
> \setuplanguage[en][patterns={en, agr}]
> ```
>
> This only works because English and Greek use a different alphabet, so there can be no conflict in the hyphenation patterns of the two languages, therefore we can load them both simultaneously. But in two languages that use the same alphabet, loading the hyphenation patterns simultaneously will necessarily lead to inappropriate hyphenation.

10.5.2 Configuring the language

ConTEXt associates the functioning of certain utilities with the specific language active at any given time. The default associations can be changed with \setuplanguage whose syntax is:

```
\setuplanguage[Language][Configuration]
```

where *Language* is the language code for the language we want to configure, and *Configuration* contains the specific configuration that we want to set (or change) for that language. Specifically, up to 32 different configuration options are allowed, but I will only deal with those that seem suitable for an introductory text such as this:

- **date**: allows us to configure the default date format. See further ahead on page 204.
- **lefthyphenmin, righthyphenmin**: the minimum number of characters that must be to the left or to the right for hyphenation of a word to be supported. For example \setuplanguage[en][lefthyphenmin=4] will not hyphenate any word if there are fewer than 4 characters to the left of the eventual hyphen.

- **spacing**: the possible values for this option are "broad" or "packed". In the first case (broad), the rules for spacing words in English will be applied, which means that after a full stop and when another character follows, a certain amount of extra blank space will be added. On the other hand, "spacing=packed" will prevent these rules from applying. For English, broad is the default.

- **leftquote, rightquote**: indicate the characters (or commands), respectively, that \quote will use to the left and right of the text that is its argument (for this command, see page 206).

- **leftquotation, rightquotation**: indicate the characters (or commands), respectively that \quotation will use to the left and right of the text that is its argument (for this command, see page 206).

10.5.3 Labels associated with particular languages

Many of ConTEXt's commands automatically generate certain texts (or *labels*), as, for example, the \placetable command that writes the label "Table xx" under the table that is inserted, or \placefigure which inserts the label "Figure xx".

These *labels* are sensitive to the language set with \mainlanguage (but not if set with \language) and we can change them with

```
\setuplabeltext[Language][Key=Label]
```

where *Key* is the term by which ConTEXt knows the label and *Label* is the text we want ConTEXt to generate. So, for example,

```
\setuplabeltext[es][figure=Imagen~]
```

would see that when the main language is Spanish, images inserted with \placefigure are not called "Figure x" but "Imagen x". Note that after the text on the label itself, a blank space must be left to ensure that the label is not attached to the next character. In the example I have used the reserved character "~"; I could also have written "[figure=Imagen{ }]" enclosing the blank space between curly brackets to ensure that ConTEXt will not get rid of it.

What labels can we redefine with \setuplabeltext? The ConTEXt documentation is not as complete as one might hope on this point. The 2013 reference manual (which is the one that explains most about this command) mentions "chapter", "table", "figure", "appendix"... and adds "other comparable text elements". We can assume that the names will be the English names of the element in question.

> One of the advantages of *free libre software* is that the source files are available to the user; so we can look into them. I have done so, and *snooping* through the source files of ConTEXt, I have discovered the file "`lang-txt.lua`", available in `tex/texmf-context/tex/context/base/mkiv` which I think is the one that contains the predefined labels and their different translations; so that if at any time ConTEXt generates a redefined text that we want to change, to see the name of the label that text is associated we can open the file in question and find that we want to change. This way we can see which label name is associated with it.

If we want to insert the text associated with a certain label somewhere in the document, we can do so with the `\labeltext` command. So, for example, if I want to refer to a table, to ensure that I name it in the same way that ConTEXt calls it in the `\placetable` command, I can write: "`Just as shown in the \labeltext{table} on the next page.`." This text, in a document where `\mainlanguage` is English, will produce: "Just as shown in the Table on the next page."

> Some of the labels redefinable with `\setuplabeltext`, are empty by default; like, for example, "`chapter`" or "`section`". This is because by default ConTEXt does not add labels to sectioning commands. If we want to change this default operation, we need only to redefine these labels in the preamble of our document and so, for example, `\setuplabeltext[chapter=Chapter~]` will see that chapters are preceded by the word "Chapter".

Finally, it is important to point out that although in general, in ConTEXt, the commands that allow several comma-separated options as an argument, the last option can end with a comma and nothing bad happens. In `\setuplabeltext` that would generate an error when compiling.

10.5.4 Some language-related commands

A. Date-related commands

ConTEXt has three date-related commands that produce their output in the active language at the time they are run. These are:

- `\currentdate`: run without arguments in a document in which the main language is English, it returns the system date in the format "Day Month Year". For example: "11 September 2020". But we can also tell it to use a different format (as would happen in the US and some other parts of the English-speaking world that follow their system of putting the month before the day, hence the infamous date, 9/11), or include the name of the day of the week (`weekday`), or include only some elements of the date (`day, month, year`)

  To indicate a different date format, "`dd`" or "`day`" represent the days, "`mm`" the months (in number format), "`month`" the months in alphabetical format in lower case, and "`MONTH`" in upper case. Regarding the year, "`yy`" will write only the last digits, while "`year`" or "`y`" will write all four. If we want some separating element between the date components, we must write it expressly. For example

```
\currentdate[weekday, dd, month]
```

when run on 9 September 2020 will write “Wednesday 9 September”.

- `\date`: this command, run without any argument, produces exactly the same output as `\currentdate`, meaning, the actual date in standard format. However, a specific date can be given as an argument. Two arguments are given for this: with the first argument we can indicate the day (“`d`”), month (“`m`”) and year (“`y`”) corresponding to the date we want to represent, while with the second argument (optional) we can indicate the format of the date to be represented. For example, if we want to know what day of the week John Lennon and Paul McCartney met, an event which, according to Wikipedia, took place on 6 July 1957, we could write

  ```
  \date[d=6, m=7, y=1957][weekday]
  ```

  and so we would find out that such an historical event happened on a Saturday.

- `\month` takes a number as an argument, and returns the name of the month corresponding to that number.

B. The `\translate` command

The translate command supports a series of phrases associated with a specific language, so that one or another will be inserted in the final document depending on the language active at any given time. In the following example, the translate command is used to associate four phrases with Spanish and English, which are saved in a memory buffer (regarding the `buffer` environment, see section 12.6):

```
\startbuffer
  \starttabulate[|*{4}{lw(.25\textwidth)|}]
    \NC \translate[es=Su carta de fecha, en=Your letter dated]
    \NC \translate[es=Su referencia, en=Your reference]
    \NC \translate[es=Nuestra referencia, en=Our reference]
    \NC \translate[es=Fecha, en=Date] \NC\NR
  \stoptabulate
\stopbuffer
```

so that if we insert the *buffer* at a point in the document where Spanish is activated, the Spanish phrases will be played, but if the point in the document where the buffer is inserted has English activated, the English phrases will be inserted. Thus:

```
\language[es]
\getbuffer
```

will generate

Su carta de fecha	Su referencia	Nuestra referencia	Fecha

while

```
\language[en]
\getbuffer
```

will generate

Your letter dated Your reference Our reference Date

C. The \quote and \quotation commands

One of the most common typographical errors in text documents occurs when quote marks (single or double) are opened but not expressly closed. To avoid this happening, ConTEXt provides the \quote and \quotation commands that will quote the text that is their argument; \quote will use single quotation marks and \quotation will use double quotation marks.

These commands are language sensitive in that they use the default character or command set for the language in question to open and close quotes (see section 10.5.2); and so, for example, if we want to use Spanish as the default style for double quotation marks – the guillemets or chevrons (angle brackets)) typical of Spanish, Italian, French, we would write:

`\setuplanguage[es][leftquotation=«, rightquotation=»]`.

These commands do not, however, manage nested quotes; although we can create the utility that does this, taking advantage of the fact that \quote and \quotation are actual applications of what ConTEXt calls *delimitedtext*, and that it is possible to define further applications with \definedelimitedtext. Thus the following example:

```
\definedelimitedtext
  [CommasLevelA]
  [left=«, right=»]

\definedelimitedtext
  [CommasLevelB]
  [left=", right="]

\definedelimitedtext
  [CommasLevelC]
  [left=`, right=']
```

will create three commands that will allow up to three different levels of quoting. The first level with side quotes, the second with double quotes and the third with single quotes.

Of course, if we are using English as our main language, then the default single and double quotation marks (curly, not straight, as you find in this document!) will be automatically used.

Chapter 11
Paragraphs, lines and vertical space

The general look of a document is determined mainly by the size and layout of the pages which we have seen in Chapter 5, by the font we have chosen, dealt with in Chapter 6, and by other matters like interline spacing, paragraph alignment and spacing between them, etc. This chapter focuses on these other matters.

11.1 Paragraphs and their characteristics

The paragraph is the fundamental unit of text for ConTEXt. There are two procedures for commencing a paragraph:

1. Inserting one or more consecutive blank lines in the source file.
2. The \par or \endgraf commands.

The first of these procedures is the one normally used since it is simpler and produces source files that are easier to read and understand. Inserting paragraph breaks with an explicit command is something usually done only inside a macro (see section 3.7.1) or in a table cell (see section 13.3).

In a well-typeset document, from a typographical point of view it is important that the paragraphs stand out visually from each other. This is usually achieved with

two procedures: by slightly indenting the first line of each paragraph or by slightly increasing the blank space between paragraphs, and sometimes by a combination of both procedures, although in some places this is not recommended because it is considered typographically redundant.

> I don't totally agree. The simple indentation of the first line does not always visually highlight the separation between paragraphs enough; but an increase in spacing not accompanied by indentation poses problems in the case of a paragraph that begins on the top of a page and we may therefore be unsure whether it is a new paragraph, or a continuation from the previous page. A combination of both procedures eliminates doubts.

Let us see, first of all how indentation of lines and paragraphs is achieved with ConTEXt.

11.1.1 Automatically indenting first lines of paragraphs

Automatic insertion of a small indent in the first line of paragraphs is disabled by default. We can enable it, disable it again and when it is enabled, indicate the extent of indentation with the `\setupindenting` command that allows the following values to indicate whether indentation should or should not be enabled:

- **`always`**: all paragraphs will be indented, regardless.
- **`yes`**: enable *normal* paragraph indentation. Certain paragraphs preceded by extra vertical spacing, such as the first paragraph of sections, or paragraphs following certain environments, will not be indented.
- **`no, not, never, none`**: disable automatic indenting of the first line in paragraphs.

In cases where we have enabled automatic indentation, we can also indicate, by means of the same command, how much indentation there should be. To do this we can expressly use a dimension (for example 1.5cm) or the symbolic words "`small`", "`medium`" and "`big`" which indicate that what we want is small, medium or big indents.

> In some typesetting traditions (among them Spanish), the default indentation was two quads. In typography, a quad (originally *quadrat) was a metal spacer used in letterpress typesetting. The term was later adopted as the generic name for two common space sizes in typography, regardless of the form of typesetting used. An em quad is a space that is one em wide; as wide as the height of the font (Wikipedia). Thus, with a 12-point letter, the quad would be 12 points wide by 12 points high. ConTEXt has two quad commands:* `\quad` *that generates one space of the kind referred to above, and* `\qquad` *that generates twice that amount, but based on the font being used. An indent of two quads with an 11 point letter will measure 22 points, and with a 12 point letter, 24 points.*

When indentation is enabled, if we don't want a certain paragraph indented we need to use the `\noindentation` command.

In general, I enable automatic indentation in my documents with `\setupindenting[yes, big]`. In this document, however, I haven't done this because if indentation were enabled, the large number of short sentences and examples would result in a visually untidy appearance of the pages.

11.1.2 Special paragraph indenting

One graphic procedure for highlighting a paragraph is to indent either the right or left (or both) sides of a paragraph. This is used, for example, for block quotes.

ConTEXt has an environment that allows us to alter paragraph indenting to highlight the text in a paragraph. This is the "`narrower`" environment:

`\startnarrower[Options] ... \stopnarrower`

where *Options* can be:

- **`left`**: indent the left margin.
- **`Num*left`**: indent the left margin, multiplying the *normal* indent by *Num* (for example `2*left`).
- **`right`**: indent the right margin.
- **`Num*right`**: indent the right margin, multiplying the *normal* indent by *Num* (for example `2*right`).
- **`middle`**: indent both margins. This is the default.
- **`Num*middle`**: indent both sides, multiplying the *normal* indent by *Num*.

When explaining the options I mentioned *normal indentation*; this refers to the amount of left and right indentation that "`narrower`" applies by default. This *amount* can be configured with `\setupnarrower` that allows the following configuration options:

- **`left`**: amount of indentation to be applied to the left margin.
- **`right`**: amount of indentation to be applied to the right margin.
- **`middle`**: amount of indentation to be applied to both margins.
- **`before`**: command to be run before entering the environment.
- **`after`**: command to be run after existing the environment.

If we want to use different configurations of the narrower environment in our document, we can assign a different name to each of them with `\definenarrower[Name] [Configuration]`

where *Name* is the name linked to this configuration and where *Configuration* allows the same values as `\setupnarrower`.

11.2 Vertical space between paragraphs

11.2.1 \setupwhitespace

As we already know from (section 4.2.2), it does not matter to ConTEXt how many consecutive blank lines there are in the source file: one or more blank lines will insert a single paragraph break in the final document. To increase the space between paragraphs, it is of no help to add an extra blank line in the source file. Instead, this function is controlled by the `\setupwhitespace` command that allows the following values:

- **`none`**: means that there will be no additional vertical space between paragraphs.
- **`small, medium, big`**: these insert, respectively, a small, medium or large vertical space. The actual size of the space inserted by these values depends on the font size.
- **`line, halfline, quarterline`**: measures the additional blank space in relation to the height of the lines and inserts an extra line, half a line, or a quarter line of space respectively.
- **`DIMENSION`**: establishes an actual dimension for the space between paragraphs. For example, `\setupwhitespace[5pt]`.

As a general rule, it is not advisable to set an exact dimension as a value for `\setupwhitespace`. It is preferable to use the symbolic values small, medium, big, line, halfline or quarterline. This is so for two reasons:

- The symbolic values are elastic dimensions (see section 3.8.2) meaning that they have *normal* dimensions but a certain decrease or increase in this value is allowed, to assist ConTEXt in typesetting pages so that paragraph breaks are aesthetically similar. But a fixed measure of separation between paragraphs makes it more difficult to achieve good pagination for the document.
- The symbolic values small, medium, big, etc., are calculated on the basis of font size, so if this changes in certain parts, it will also change the amount of vertical spacing between paragraphs, and the end result will always be harmonious. Conversely, a fixed value for vertical spacing will not be affected by changes in font size, which will normally translate into a document with poorly distributed white space (from the aesthetic point of view) and not in accordance with the rules of typographical adjustment.

When a value has been set for vertical paragraph spacing, two additional commands are available: `\nowhitespace`, which eliminates any extra space between

particular paragraphs, and \whitespace which does the opposite. However, these commands are rarely needed, because the fact is that ConTEXt manages the vertical spacing between paragraphs quite well on its own; especially if one of the predefined dimensions has been inserted as a value, calculated from the current active font size and spacing.

The meaning of \nowhitespace is obvious. But not \whitespace itself, necessarily, because what is the point of ordering vertical spacing for particular paragraphs given that vertical spacing has already been generally established for all paragraphs? However, when writing advanced macros, \whitespace can be useful in the context of a loop that has to make a decision based on the value of a certain condition. This is more or less advanced programming, and I won't go into it here.

11.2.2 Paragraphs with no extra vertical space between them

If we want particular parts of our document to have paragraphs that are not separated by extra vertical space, we can of course, modify the general configuration of \setupwhitespace, but that is, in a way, contrary to the ConTEXt philosophy in which the general configuration commands should be placed exclusively in the preamble of the source file, so as to achieve a consistent and easily amendable general appearance for documents. Hence the "packed" environment, whose general syntax is

```
\startpacked[Space] ... \stoppacked
```

where *Space* is an optional argument indicating what amount of vertical space is desired between the paragraphs in the environment. If omitted, no extra vertical space will be applied.

11.2.3 Adding additional vertical space at a particular point in the document

If, at a particular point in the document the normal vertical spacing between paragraphs is not sufficient, we can use the \blank command. Used without arguments, \blank will insert the same amount of vertical space as has been set with \setupwhitespace. But we can indicate either a specific dimension between square brackets, or one of the symbolic values calculated from the font size: small, medium or big. We can also multiply those sizes by some whole number, and so on, for example, \blank[3*medium] will insert the equivalent of three medium line breaks. We can also put two sizes together. For example, \blank[2*big, medium] will insert two large and a medium break.

Since \blank is designed to increase the vertical space between paragraphs, it has no effect if a page break is inserted between the two paragraphs whose spacing

should be increased; and if we insert two or more \blank commands in succession, only one of them will apply (the one with the most space to be inserted). Nor does a \blank command placed after a page break have any effect. However, in these cases we can force the insertion of vertical spacing using the symbolic word "force" as a command option. So, for example, if we want the chapter titles in our document to appear further down the page, so that the total length of the page is less than the rest of the pages (a relatively frequent typographical practice), we must write in the configuration of \chapter command, for example:

```
\setuphead
  [chapter]
  [
    page=yes,
    before={\blank[4cm, force]},
    after={\blank[3*medium]}
  ]
```

This sequence of commands will ensure that chapters always start on a new page and that the chapter label moves four centimetres downwards. Without using the "force" option, this will not work.

11.2.4 \setupblank and \defineblank

Earlier, I said that \blank, used without arguments, is equivalent to \blank[big]. However, we can change this with \setupblank, setting it as \setupblank[0.5cm] for example, or \setupblank[medium]. Used without arguments, \setupblank will adjust the value to the size of the current font.

Just the same as with \setupwhitespace the white space inserted by \blank, when its value is one of the predefined symbolic values, is an elastic dimension that allows for some adjustment. We can change this with "fixed", with the possibility, later on, of restoring the default value with ("flexible"). Thus, for example, for text in double columns it is recommended to set \setupblank[fixed, line], and when going back to a single column \setupblank[flexible, default].

With \defineblank we can associate a certain configuration with a name. The general format of this command is:

```
\defineblank[Name] [Configuration]
```

Once our white space configuration is defined, we can use it with \blank[ConfigurationName].

11.2.5 Other procedures for achieving more vertical space

In TEX the command that inserts extra vertical space is `\vskip`. This command, like almost all TEX commands, also works in ConTEXt but its use is strongly advised against since it interferes with the internal functioning of some of ConTEXt's macros. In its place it is suggested to use `\godown` whose syntax is:

```
\godown[Dimension]
```

where *Dimension* needs to be a number with or without decimals, followed by a unit of measure. For example, `\godown[5cm]` will shift 5 centimetres down on the page; although if change of page is less than this amount, `\godown` will only move to the next page. Similarly, `\godown` will have no effect at the beginning of a page, although we can *trick it* by writing, for example "`\␣\godown[3cm]`"[1] that will first insert a blank space that will mean we are no longer at the beginning of the page, and will then go down three centimetres.

> As we know, `\blank` also allows a precise dimension as an argument. Therefore, from the user's point of view, writing `\blank[3cm]` or `\godown[3cm]` is practically the same. However, there are some subtle differences between them. So, for example, two consecutive `\blank` commands cannot be accumulated and when this happens, only the one that imposes a greater distance is applied. Two or more `\godown` commands, on the other hand, can accumulate perfectly.

Another rather useful TEX command, the use of which poses no problems in ConTEXt, is `\vfill`. This command inserts a flexible vertical blank space going as far as the bottom of the page. It is as if the command *pushes* down what is written after it. This allows for interesting effects such as how to place a certain paragraph at the bottom of the page, by simply preceding it with `\vfill`. Now, the effect of `\vfill` is difficult to appreciate if its use is not combined with forced page breaks, because there is little point in pushing a paragraph or line of text down if the paragraph, as it grows, grows upwards.

So, for example, to ensure that a line is placed at the bottom of the page, we should write:

```
\vfill
Line at the bottom
\page[yes]
```

Like all other commands that insert vertical space, `\vfill` has no effect at the beginning of a page. But we can *trick it* by preceding it with a forced blank space. So, for example:

---

[1] Recall that we are using the '␣' character in this document to represent a blank space when it is important for us to see it.

```
\page[yes]
\ \vfill
Centre line
\vfill
\page[yes]
```

will vertically centre the phrase “centre line” on the page.

11.3 How ConTEXt builds lines that form paragraphs

One of the main duties of a typesetting system is to take a long string of words and divide it into individual lines of the appropriate size. For example, each paragraph in this text has been divided into lines 15 centimetres wide, but the author has not had to worry about such details, as ConTEXt chooses the breakpoints after considering each paragraph in its entirety, so that the final words of a paragraph can really influence the division of the first line. As a result, the space between the words in the entire paragraph is as uniform as possible.

> This is one aspect where we can best note the different way word processors work and the better quality obtained with systems such as ConTEXt. Because a word processor, when it reaches the end of the line and jumps to the next, adjusts the white space in the line just finished to enable right justification. It does this with each line, and at the end, each line in the paragraph will have different interword spacing. This can cause a very bad effect (e.g. ‘rivers’ of white space running through a text). ConTEXt, on the other hand, processes the paragraph in its entirety and for each line calculates how many breakpoints are admissible and the amount of interword spacing that would result from a line break. As the breakpoint of a line affects the potential breakpoints of the next lines, the total number of possibilities can be very high; but that is not a problem for ConTEXt. It will make a final decision based on the entire paragraph, ensuring that the space between words on each line is *as similar as possible*, which results in much better typeset paragraphs; visually more compact.

To do this, ConTEXt tests different alternatives, and assigns a *badness* value to each of them based on its parameters. These were established after an in-depth study of the art of typography. Finally, after having explored all possibilities, ConTEXt chooses the least unsuitable option (the one with the least badness value). In general, this functions quite well, but there will inevitably be cases where line breakpoints are chosen that are not the best, or that do not appear to us to be the best. Therefore, sometimes we will want to tell the program that certain places are not good breakpoints. Then on other occasions we will want to force a break at a particular point.

11.3.1 Use of the reserved ‘~’ character

The main candidates for line breakpoints are obviously the white space between words. To indicate that a certain space should never be replaced by a line break,

we use, as we already know, the '~' reserved character, which TEX calls a *tie*, tying two words together.

It is generally recommended to use this non-breaking space in the following cases:

- Between the parts that make up an abbreviation. For example, `U~S`.
- Between abbreviations and the term they refer to. For example, `Dr~Anne Ruben` or `p.~45`.
- Between numbers and the term that goes with them. For example, `Elizabeth~II`, `45~volumes`.
- Between digits and the symbols preceding or following them so long as they are not superscripts. For example, `73~km`, `$~53`; however, `35'`.
- In percentages expressed in words. For example, `twenty~per~cent`.
- In groups of numbers separated by white space. For example, `5~357~891`. Although in these cases it is preferable to use what is called *thin space* achieved in ConTEXt with the `\,` command, and therefore write `5\,357\,891`.
- To avoid an abbreviation being the only item on that line. For example:

```
There are sectors such as entertainment, communications media,
commerce,~etc.
```

To these cases, KNUTH (the father of TEX) adds the following recommendations:

- After an abbreviation that is not at the end of a sentence.
- In reference to parts of a document such as chapters, appendices, figures, etc. For example `Chapter~12`.
- Between the first name and the initial of the second name of a person, or between the initial of the first name and the surname. For example, `Donald~E. Knuth`, `A.~Einstein`.
- Between mathematical symbols in apposition to names. For example, `dimension~$d$`, `width~$w$`.
- Between symbols in series. For example `{1,~2, \dots,~$n$}`.
- When a number is strictly bound up with a preposition. For example `from 0 to~1`.
- When mathematical symbols are expressed in words. For example, `equals~a~$n$`.
- In lists within a paragraph. For example: `(1)~green, (2)~red, (3)~blue`.

Many cases? Without a doubt, typographic perfection has a cost in terms of extra effort. It is clear that if we don't want to, we don't have to apply these rules, but it doesn't hurt to know them. Besides – and here I speak from experience – once we get used to applying them (or any of them), doing so becomes automatic. It is like putting accents on words as we write them (as we need to do in Spanish): for those of us who do, if we are used to writing them automatically, it doesn't take

us any longer to write a word with an accent than it would for a word without an accent.

11.3.2 Word hyphenation

Except for languages made up mostly of monosyllables, it is quite difficult to get an optimal result if line breakpoints are only in the space between words. Hence ConTEXt also analyses the possibility of inserting a line break between two syllables of a word; and to do this it is essential for it to know the language the text is in, since hyphenation rules are different for each language. Thus the importance of the `\mainlanguage` command in the document preamble.

It can happen that ConTEXt has been unable to hyphenate a word suitably. Sometimes this can be because its own rules for splitting words get in the way of the task (for example, ConTEXt never splits a word into two parts if these parts do not have a minimum number of letters); or because the word is ambiguous. After all, what might ConTEXt do with the word "unionised"? The word could appear in a phrase like "the unionised workforce", but it could also appear in a chemistry text as "an unionised particle" (i.e. un-ionised). And what if ConTEXt had to deal with the word "manslaughter" as the last word on a page before a page break. It may split the word as man-slaughter (correct) but it may also split it as mans-laughter (ambiguous).

Whatever the reason, if we are not satisfied with how a word has been split, or it is incorrect, we can change it by expressly indicating the potential points where a word can be split with the `\-` control symbol. So, for example, if "unionised" gave us any problems, we could write it in the source file as "`union\-ised`"; or if we had a problem with "manslaughter", we could write "`man\-slaughter`".

If the problem word is used several times in our document then the preference is to show how it should be hyphenated in our preamble with the `\hyphenation` command: this command, which is intended to be included in the preamble of the source file, takes one or more words (commas-separated) as an argument, indicating the points at which they can be split with a hyphen. For example:

```
\hyphenation{union-ised, man-slaughter}
```

If the word that is the subject of this command does not contain a hyphen, the effect will be that the word will never be hyphenated. This same effect can be achieved by using the `\hbox` command that creates an indivisible horizontal box around the word, or `\unhyphenated` that prevents the hyphenation of the word or words it takes as arguments. But while `\hyphenation` acts globally, `\hbox` and `\unhyphenated` act locally, meaning that the `\hyphenation` command affects all occurrences in the document of words included in its argument; unlike `\hbox`

or \unhyphenated that only act at the point in the source file where they are encountered.

> Internally, how hyphenation works is controlled by the TEX \pretolerance and \tolerance variables. The first of these controls the admissibility of a split done only on white space. By default it is 100, but if we alter it, for example, to 10 000, then ConTEXt will always consider it acceptable for there to be a line break that does not mean splitting words according to syllables, meaning that *de facto*, we are removing hyphenation based on syllables. While if, for example, we were to set the \pretolerance value to a -1, we would be forcing ConTEXt to use word hyphenation at the end of the line every time.
>
> We can directly set an arbitrary value for \pretolerance by simply assigning a value there in our document. For example:
>
> ```
> \pretolerance=10000
> ```
>
> but we can also manipulate this value with the "lesshyphenation" and "morehyphenation" values in \setupalign. In this regard see section 11.6.1.

11.3.3 Tolerance level for line breaks

When looking for possible line break points, ConTEXt is usually quite strict, which means that it prefers to allow a word to go beyond the right-hand margin because it has not been able to hyphenate it, and prefers not to insert a line break before the word if this results in too great an increase in interword space on that line. This default behaviour normally provides optimal results, and only exceptionally do certain lines stand out somewhat on the right-hand side. The idea is that the author (or typesetter), reviews these exceptional cases once the document is finished, to make the appropriate decision, which could be a \break command before the word that extends beyond, or could also mean wording the paragraph differently so that this word shifts position elsewhere.

However, in some cases, ConTEXt's low tolerance can be a problem. In these cases we can tell it to be more tolerant with white space in lines. We have the \setuptolerance command for this, allowing us to alter the level of tolerance in calculating line breaks, which ConTEXt calls "horizontal tolerance" (because it affects horizontal space) and "vertical tolerance" when calculating page breaks. We will talk about this in section 11.6.2.

Horizontal tolerance (which is the one that effects line breaks), is set at the "verystrict" value by default. We can alter this by setting, as alternatives, any of the following values: "strict", "tolerant", "verytolerant" or "stretch". So, for example,

```
\setuptolerance[horizontal, verytolerant]
```

will make it almost impossible for a line to go beyond the right-hand margin, even if this means establishing a very large and unsightly spacing between words on a line.

11.3.4 Forcing a line break at a certain point

To force a line break at a certain point we use the `\break`, `\crlf` or \\ commands. The first of these, `\break`, enters a line break at the point where it is located. This will most probably cause the line where the command is placed to be aesthetically deformed, with an immense amount of white space between the words on that line. As can be seen in the following example in which the `\break` command in the third line (of the source fragment on the left) results in a second quite ugly line (in the formatted text on the right).

```
On the corner of the old quarter I saw
him \emph{swagger} along like
the\break tough guys do when they walk,
hands always in their overcoat pockets,
so no one can know which of them carries
the dagger.
```

On the corner of the old quarter I saw him *swagger* along like the
tough guys do when they walk, hands always in their overcoat pockets, so no one can know which of them carries the dagger.

To avoid this effect, we can use the \\ or `\crlf` commands that also insert a forced line break, but they fill in the original line with enough blank space to align it to the left:

```
On the corner of the old quarter I saw
him \emph{swagger} along like
the\\ tough guys do when they walk,
hands always in their overcoat pockets,
so no one can know which of them carries
the dagger.
```

On the corner of the old quarter I saw him *swagger* along like the
tough guys do when they walk, hands always in their overcoat pockets, so no one can know which of them carries the dagger.

On *normal* lines, as far as I know, there are no differences between \\ and `\crlf`; but in a section title there is a difference:

- \\ generates a line break in the body of the document, but not when the section title is transferred to the table of contents.
- `\crlf` generates a line break that is applied both in the body of the document and when the section title is transferred to the table of contents.

A line break should not be confused with a paragraph break. A line break simply ends the current line and starts the next line, but keeps us in the same paragraph, so the separation between the original line and the new line will be determined by the normal spacing within a paragraph. Therefore, there are only three scenarios in which it may be recommended to force a line break:

- In very exceptional cases when ConTEXt has not been able to find a suitable line break, so that a line protrudes on the right. In these cases (which occur very rarely, mainly when the line has indivisible *boxes*, or *verbatim* text [see section 10.2.3]), it could be helpful to force a line break with `\break` just before the word that protrudes into the right margin.

- In paragraphs that are actually made up of individual lines, each with information independent of that of the previous lines, for example, the heading of a letter in which the first line may contain the name of the sender, the second the recipient, and the third the date; or in a text talking about the authorship of a work, where one line has the author's name, another their office or academic position and perhaps a third line with the date, etc. In these cases the line break should be forced with the \\ or `\crlf` commands. It is also common for these kinds of paragraphs to be right-aligned.

- When writing poems or similar kinds of texts, to separate one verse from another. Although in this latter case it is preferable to use the `lines` environment explained in section 11.5.1.

11.4 Interline space

Interline space is the distance separating the lines that make up a paragraph. ConTEXt calculates this automatically on the basis of the actual font being used, and, above all, on the base size set with `\setupbodyfont` or `\switchtobodyfont`.

We can influence interline space with the `\setupinterlinespace` command that allows for three different kinds of syntax:

- `\setupinterlinespace [..Interline space..]`, where *Interline space* is a precise value or a symbolic word that assigns a predefined interline space:

  – When it is a precise value it can be a dimension (for example, 15pt), or a simple, whole or decimal number (for example, 1.2). In this latter case the number is interpreted as “number of lines” based on ConTEXt's default interline space.

  – When it is a symbolic word this can be “`small`”, “`medium`” or “`big`”, each of which applies a small, medium or big interline space respectively, always based on the default interline space ConTEXt would apply.

- `\setupinterlinespace [..,..=..,..]`. In this mode, the interline space is set by explicitly altering the based measures with which ConTEXt calculates the appropriate interline spacing. In this mode the spacing is set by explicitly altering the measures on the basis of which ConTEXt calculates the appropriate

spacing. I have previously said that line spacing is calculated on the basis of the specific font and its size; but that was to keep things very simple: actually what the font and its size do is to establish certain measures on the basis of which the interline space is calculated. By means of this `\setupinterlinespace` approach, these measures are modified and therefore, so is the interline space. The actual measures and values that can be manipulated by this procedure (the meaning of which I will not explain because it goes beyond the scope of a simple *introduction*), are: `line`, `height`, `depth`, `minheight`, `mindepth`, `distance`, `top`, `bottom`, `stretch` and `shrink`.

- `\setupinterlinespace [Name]`. With this mode, we establish or configure a specific and customised type of line spacing previously defined with `\defineinterlinespace`.

With

```
\defineinterlinespace[Name] [Configuration]
```

we can associate a certain interline space configuration with a specific name that we can then simply trigger at some point in our document with `\setupinterlinespace[Name]`. To return to normal interline space, we would then need to write `\setupinterlinespace[reset]`.

11.5 Other matters relating to lines

11.5.1 Converting line breaks in the source file into line breaks in the final document

As we already know (see section 4.2.2), by default ConTEXt ignores the line breaks in the source file that it considers to be simple blank spaces, unless there are two or more consecutive line breaks, in which case a paragraph break will be inserted. However, there may be some situations in which we are interested in respecting the line breaks of the original source file as they were put there, for example, when writing poetry. For this, ConTEXt offers us the "`lines`" environment whose format is:

```
\startlines[Options] ... \stoplines
```

where the options can be any of the following, amongst others:

- `space`: When this option is set with the "`on`" value, in addition to respecting the line breaks in the source file, the environment will also respect blank spaces in the source file, temporarily ignoring the absorption rule.
- `before`: Text or command to run before entering the environment.

- **after**: Text or command to run after exiting the environment.
- **inbetween**: Text or command to run when entering the environment.
- **indenting**: Value indicating whether or not to indent paragraphs in the environment (see section 11.1.1).
- **align**: Alignment of lines in the environment (see section 11.6).
- **style**: Style command to apply within the environment.
- **color**: Colour to apply within the environment.

So, for example,

```
\startlines
  One-one was a race horse.
  Two-two was one too.
  One-one won one race.
  Two-two won one too.
\stoplines
```

One-one was a race horse.
Two-two was one too.
One-one won one race.
Two-two won one too.

We can also modify the default way the environment works with `\setuplines` and, as with so many of ConTEXt's commands, it is also possible to assign a name to a particular configuration of this environment. We do this with the `\definelines` command whose syntax is:

```
\definelines[Name] [Configuration]
```

where, as a configuration, we can include the same options that have been explained generally for the environment. Once we have defined our customised line environment, to insert it we should write:

```
\startlines[Name] ... \stoplines
```

11.5.2 Line numbering

In certain kinds of texts it is common to establish some kind of line numbering, for example, in texts on computer programming where it is relatively common for the code fragments offered as examples to have their lines numbered, or in poems, critical editions, etc. For all these situations ConTEXt offers the `linenumbering` environment whose format is

```
\startlinenumbering[Options] ... \stoplinenumbering
```

The available options are:

- **continue**: In cases where there is more than one part of our document requiring lines to be numbered, this option sees that the numbering restarts for each part ("continue=no", the default value). On the other hand, if line numbering is meant to continue on from where the previous part left off, we choose "continue=yes".

- **start**: Indicates the number of the first line in cases where we do not want it to be '1', or for it to correspond with the previous enumeration.

- **step**: All the lines included in the environment will be numbered, but, by means of this option, we can indicate that the number is printed only at certain intervals. With poems, for example, it is common that the number only appears in multiples of 5 (verses 5, 10, 15...).

All these options can be indicated, in general for all the *linenumbering* environments in our document, with \setuplinenumbering. This command also allows us to configure other aspects of line numbering:

- **conversion**: Line numbering type. It can be any of the ones explained on page 136 regarding the numbering of chapters and sections.

- **style**: Command (or commands) determining the style the line numbering will have (font, size, variant...).

- **color**: Colour the line number will be printed in.

- **location**: Where the line number will be placed. It can be any of the following: text, begin, end, default, left, right, inner, outer, inleft, inright, margin, inmargin.

- **distance**: Distance between the line number and the line itself.

- **align**: Number alignment. Can be: inner, outer, flushleft, flushright, left, right, middle or auto.

- **command**: Command to which the line number will be passed as a parameter before printing.

- **width**: Width reserved for printing the line number.

- **left, right, margin**:

We can also create different customised line numbering configurations with \definelinenumbering such that the configuration be be associated with a name:

```
\definelinenumbering[Name] [Configuration]
```

Once a specific configuration has been defined and associated with a name, we can use it with

```
\startlinenumbering[Name] ... \stoplinenumbering
```

11.6 Horizontal and vertical alignment

The command that controls text alignment in general is `\setupalign`. This command is used to control both horizontal and vertical alignment.

11.6.1 Horizontal alignment

When the *exact* width of a line of text does not take up all the width possible, this poses a problem of what to do with the resulting white space.[1] We can basically do three things in this regard:

1. Accumulate it on one of the two sides of the line: if we accumulate it on the left hand side, the line will look *a little pushed* to the right, while if we accumulate it on the right hand side the line remains on the left hand side. We are talking, in the former case, about *right alignment* and, in the latter, about *left alignment*. By default, ConTEXt applies left alignment to the last line of paragraphs.

   When several consecutive lines are aligned on the left, the right hand side is irregular; but when the alignment is on the right, the side that looks uneven is the left. To name the options that align one or other side, ConTEXt does not set the side where they are aligned, but the side where they are uneven. Therefore, the `flushright` option results in left alignment and `flushleft` in right alignment. As abbreviations of `flushright` and `flushleft`, `\setupalign` also supports `right` and `left` as values. But **attention**: here the meaning of the words is deceptive. Even though *left* means "left" and *right* means "right", `\setupalign[left]` aligns on the right and `\setupalign[right]` aligns on the left. In case the reader wonders why this comment has been made, it would be worth quoting from the ConTEXt wiki: "ConTeXt uses flushleft and flushright options. The right and left alignments are backwards from the usual directions in all commands that accept an alignment option, in the sense of 'ragged left' and 'ragged right'. Unfortunately, when Hans was first writing this part of ConTeXt, he was thinking of 'ragged right' and 'ragged left' alignment, rather than 'flush left' and 'flush right'. And now that it's been this way a while, it's impossible to change it, because changing it would break backward compatibility with all of the existing documents that use it."

[1] By *exact* width I mean the width of the line *before* ConTEXt adjusts the size of interword space to enable justification.

In documents prepared for double-sided printing, in addition to the right and left margins, there are also inner and outer margins. The values `flushinner` (or simply `inner`) and `flushouter` (or simply `outer`) establish the corresponding alignment in these cases.

2. Distribute it across both margins. The result will be that the line is centred. The `\setupalign` option that does this is `middle`.

3. Distribute it among all the words making up the line,if necessary by increasing interword space, so that the line becomes exactly the same width as the space available to it. In these cases we talk about *justified lines*. This is also ConTEXt's default value which is why there is no special option in `\setupalign` to establish it. However, if we have altered alignment justified by default, we can restore it with `\setupalign[reset]`.

The value for `\setupalign` that we have just seen (`right, flushright, left, flushleft, inner, flushinner, outer, flushouter` and `middle`) can be combined with `broad`, which results in somewhat rougher alignment.

> Two other possible values of `\setupalign` that affect the horizontal alignment, have to do with the hyphenation of words at the end of the line, because whether this is done or not depends on whether the exact measure of the line is larger or smaller; which in turn affects the remaining white space.
>
> To this effect, `\setupalign` allows the `morehyphenation` value which makes ConTEXt work harder to find breakpoints based on hyphenation, and `lesshyphenation` which produces the opposite effect. With `\setupalign[horizontal, morehyphenation]`, the remaining white space in the lines will be reduced and therefore the alignment will be less apparent. On the contrary, with `\setupalign[horizontal, lesshyphenation]`, there will be more white space left, and the alignment will be more visible.

`\setupalign` is intended to be included in the preamble and affect the whole document or, to be included at a specific point and affect everything from that point to the end. If we only want to change the alignment of one or several lines we can use:

- The "`alignment`" environment, intended to affect several lines. Its general format is:

  ```
  \startalignment[Options] ... \stopalignment
  ```

  where *Options* are any of those allowable for `\setupalign`.

- The `\leftaligned`, `\midaligned` or `\rightaligned` commands that cause left, centred or right alignment respectively; and if we want the last word in a paragraph (but only this and not the rest of the line) to be right aligned we can use `\wordright`. All these commands require the text to be affected to be between curly brackets.

Note, on the other hand, that if the words "`right`" and "`left`" in `\setupalign` cause the opposite alignment to what the name suggests, the same does not happen with the `\leftaligned` and `\rightaligned` commands that bring about exactly the kind of alignment that their name suggests: `left` on the left, and `right` on the right.

11.6.2 Vertical alignment

If horizontal alignment comes into play when the width of a line does not take up all the space available to it, vertical alignment affects the height of the whole page: if the *exact* text height of a page does not take up all the height available to it, what do we do with the remaining white space? We can pile it up at the top ("`height`"), which means that the text on the page will be pushed down; we can pile it up at the bottom ("`bottom`") or distribute it among the paragraphs ("`line`"). The default value for vertical alignment is "`bottom`".

Vertical level of tolerance

In the same way we can alter ConTEXt's level of tolerance with regard to the amount of horizontal space permissible in a line (horizontal tolerance) with `\setuptolerance`, we can also alter its vertical tolerance, i.e. tolerance for space between paragraphs larger than what ConTEXt, by default, considers reasonable for a well-typeset page. The values possible for vertical tolerance are the same as for horizontal tolerance: `verystrict, strict, tolerant` and `verytolerant`. The default value is `\setuptolerance [vertical, strict]`.

Controlling widows and orphans

One aspect that indirectly affects vertical alignment is the control of widows and orphans. Both phenomena imply that a page break causes one line of a paragraph to be isolated on a different page from the rest of the paragraph. This is not considered to be typographically appropriate. If the line that is separated from the rest of the paragraph is the first one on the page, we are talking about a *widowed line*; if the line separated from its paragraph is the last one on the page then we are talking about an *orphaned line*.

By default, ConTEXt does not implement a control to ensure these lines do not occur. But we can change this by altering some of ConTEXt's internal variables: `\widowpenalty` controls widowed lines and `\clubpenalty` controls orphaned lines. Thus, the following statements in the preamble to our document will ensure that this control is carried out:

```
\widowpenalty=10000
\clubpenalty=10000
```

Carrying out this control means that ConTEXt will avoid inserting a page break that separates the first or last line of a paragraph from the page on which the rest

is found. This avoidance will be more or less rigorous depending on the value we assign to the variables. With a value of 10 000, like the one I used in the example, the control will be absolute; with a value of, for example, 150, the control will not be as rigorous and occasionally there may be some widowed or orphaned lines when the alternative is worse in typographical terms.

Chapter 12
Special constructions and paragraphs

12.1 Footnotes and endnotes

Notes are "secondary textual elements employed for various purposes, such as clarifying or extending the main text, providing the bibliographic reference for the sources, including citations, referring to other documents or stating the meaning of the text" [*Libro de Estilo de la Lengua española* (Spanish Language Style Guide), p. 195]. They are particularly important in texts of an academic nature. They can be placed at different points on the page or in the document. Today, the most widespread ones are those located at the foot of the page (called, therefore, footnotes); sometimes they are also located in one of the margins (margin notes), at the end of each chapter or section, or at the end of the document (endnotes). In particularly complex documents, there may also be different *series* of notes: author's notes, translator's notes, updates, etc. In particular, in critical editions the note apparatus can become rather complex and only a few typesetting systems are capable of supporting it. ConTEXt is one of these. There are numerous commands available to establish and configure different types of notes.

To explain this, it is useful to start by pointing out the various elements that can be involved in a note:

- *Mark* or note *anchor*: The sign placed in the body of the text to indicate that there is a note linked to it. Not all types of notes have an *anchor* associated with them, but when there is one, this *anchor* appears in two places: at the point in the main text to which the note refers, and at the beginning of the note text itself. The presence of the same reference mark in both places is what allows the note to be associated with the main text.

- The note *ID or identifier*: The letter, number or symbol that identifies the note and distinguishes it from other notes. Some notes, for example margin notes, can lack an ID. When this is not the case, the ID normally coincides with the *anchor*.

  > If we think exclusively of footnotes, we will see no difference between what I have just called a *reference mark* and the *id*. We clearly see the difference in other kinds of notes: Line notes, for example, have an id, but not a reference mark.

- *Text* or *contents* of the note, always located at a different point on the page or in the document than the command that generates the note and indicates its content.

- *Label* associated with the note: A label or name associated with a note that is not shown in the final document, but allows us to refer to it and retrieve its ID elsewhere in the document.

12.1.1 Types of notes in ConTEXt and commands associated with them

We have various types of notes in ConTEXt. For the moment I will only list them, describing them in general terms and providing information about the commands that generate them. Later I will develop the first two:

- **Footnotes:** Undoubtedly the most popular, to the extent that it is common for all types of notes to be referred to generically as *footnotes*. Footnotes introduce a *mark* with the note's *id* at the point in the document where the command is found, and insert the text of the note itself at the bottom of the page where the mark appears. They are created with the `\footnote` command.

- **Endnotes:** These notes, which are created with the command `\endnote`, are inserted at the point in the document where a mark with the note's ID is found; but the note's contents are inserted at another point in the document, and the insertion is produced by a different command (`\placenotes`).

- **Margin notes:** As their name suggests, they are written in the margin of the text and there is no ID or automatically generated mark or anchor in the body

of the document. The two main commands (but not the only ones) that create them are `\inmargin` and `\margintext`.

- **Line notes:** A type of note typical of environments where lines are numbered, such as in the case of `\startlinenumbering ... \stoplinenumbering` (see section 11.5.2). The note, which is usually written at the bottom, refers to a specific line number. They are generated with the `\linenote` command which is configured with `\setuplinenote`. This command prints no *mark* in the body of the text, but in the note itself it prints the line number the note refers to (used as the *ID*).

I will now exclusively develop the first two types of notes:

- Margin notes are treated elsewhere (section 5.7).
- Line notes have a highly specialised use (especially in critical editions) and I believe that in an introductory document like this one, it is enough for the reader to know that they exist.

> However, for the interested reader I recommend a video (in Spanish) accompanied by a text (also in Spanish) about critical editions in ConTEXt, the author of which is Pablo Rodríguez. It is available at this link. It is also quite useful for understanding several of the general settings of notes in general.

12.1.2 A close look at footnotes and endnotes

The syntax for the footnotes and endnotes commands and the configuration and customisation mechanisms they have are quite similar, since, in reality, both types of notes are particular instances of a more general construction (notes), other instances of which can be set with the `\definenote` command (see section 12.1.4).

The syntax of the command that creates each of these kinds of notes is as follows:

```
\footnote[Label]{Text}
\endnote[Label]{Text}
```

where

- *Label* is an optional argument that assigns the note a label that will allow us to refer to it elsewhere in the document.
- *Text* is the content of the note. It can be as long as we wish, and include special paragraphs and settings, although it should be noted that when it comes to footnotes, correct page layout is quite difficult in documents with abundant and excessively long notes.

> In principle, any command that could be used in the main text can be used in the note text. However, I have been able to verify that certain constructions and characters that do

not pose any kind of problem in the main text, do generate a compilation error when they take place in the note text. These cases I found as I was testing, but I have not organised them in any way.

When the *Label* argument has been used to set a label for the note, the \note command allows us to retrieve the ID of the note in question. This command prints the ID of the note associated with the label it takes as an argument on the document. Thus, for example:

```
Humpty Dumpty\footnote[humpty]{Probably the
best-known English nursery rhyme character}
sat on a wall, Humpty Dumpty\note[humpty]
had a great fall.\\
All the king's horses and
all the king's men Couldn't put
Humpty\note[humpty] together again
```

Humpty Dumpty[1] sat on a wall,
Humpty Dumpty[1] had a great fall.
All the king's horses and all the king's men
Couldn't put Humpty[1] together again

[1] Probably the best-known English nursery rhyme character

The main difference between \footnote and \endnote is the place where the note appears:

\footnote As a rule, it prints the note text at the bottom of the page on which the command is located, so that the note mark and its text (or the beginning of the text, if it is to be spread over two pages) will appear on the same page. To do this, ConTEXt will make the necessary adjustments in typesetting the page by calculating the space required by the location of the note at the bottom of the page.

But in some environments, \footnote will insert the text of the note, not at the bottom of the page itself but beneath the environment. This is the case, for example, in tables, or in the columns environment. In these cases, if we want the notes inside the environment to be located at the bottom of the page, instead of \footnote the command we should use is \footnotetext in combination with the \note command mentioned above. The former, which also supports a label as an optional argument, prints only the note text but not the mark. But as \note prints only the mark without the text, the combination of both allows us to place the note at the point where we want it. So, for example, we could write \note[MyLabel] within a table or a multi-column environment, and then, once out of that environment, \footnotetext[MyLabel]{Note text}.

Another example of the use of \footnotext in combination with \note would be notes inside other notes. For example:

```
This%
\footnote{or this\note[noteB], if you prefer.}%
\footnotetext[noteB]
{or possibly even this one\note[noteC].}
\footnotetext[noteC]{could be something
entirely different.}
is a sentence with nested notes.
```

This[1] is a sentence with nested notes.

[1] or this[2], if you prefer.
[2] or possibly even this[3].
[3] could be something entirely different.

\endnote only prints the note anchor at the point in the source file where it is located. The actual content of the note is inserted at another point in the document with another command, (\placenotes[endnote]) which, at the point where it is located, will insert the contents of *all* the endnotes of the document (or of the chapter or section in question).

12.1.3 Local notes

The \startlocalfootnotes environment means that the footnotes included within it are considered to be *local* notes, which means that their numbering will be reset and that the content of the notes will not be automatically inserted along with the rest of the notes, but only at the point in the document where the \placelocalfootnotes command is found, which may or may not be within the environment.

12.1.4 Creating and using customised types of notes

We can create special types of notes with the \definenote command. This can be useful in complex documents where there are notes from different authors, or for different purposes, to graphically distinguish each of the types of notes in our document by means of a different format and different numbering.

The syntax of \definenote is as follows:

```
\definenote[Name][Model][Configuration]
```

where

- *Name* is the name we assign to our new type of note.
- *Model* is the note model that will be used initially. It can be `footnote` or `endnote`; in the former case our note model will work as footnotes, and in the latter case as endnotes, although to insert them in the document we would not use \placenotes[endnote] but \placenotes[Name] (the name we have assigned to these kinds of notes).

In theory this argument is optional, although in my tests some notes created without it were not visible, and I have not had the patience to find out what the cause was.

- *Configuration* is an optional second argument that allows us to distinguish our new type of notes from its model: either by setting a different format, or a different type of numbering, or both.

  According to the official list of ConTEXt commands (see section 3.6) the settings that can be provided when the new type of note is created are based on those that could be provided later with \setupnote. However, as we shall see shortly, there are actually two possible commands for setting up notes: \setupnote and \setupnotation. So I think it is preferable to omit this argument when creating the note type, and then set up our new notes using the appropriate commands. At least this is easier to explain.

For example, the following item will create a new note type called "BlueNote" that will be similar to footnotes but its contents will be printed in bold and blue:

```
\definenote  [BlueNote] [footnote]
\setupnotation
  [BlueNote]
  [color=blue, style=bf]
```

Once we have created a new note type, e.g. *BlueNote* the command allowing us to use it will be available. In our example this will be \BlueNote the syntax of which will be similar to \footnote:

```
\BlueNote[Label]{Text}
```

12.1.5 Configuring notes

The configuration of notes (footnotes or endnotes, notes created with \definenote and also line notes set up with \linenote) is achieved with two commands: \setupnote and \setupnotation[1]. The syntax for both is similar:

```
\setupnote[NoteType][Configuration]
```

[1] \setupnote has 35 *direct* configuration options and 45 additional options inherited from \setupframed; \setupnotation has 45 direct configuration options and another 23 inherited from \setupcounter. Since these options are not documented and, although for many of them we can work out their usefulness from their name, we need to check whether our intuition is true or not; and also taking into account that many of these options allow a number of values and they all have to be tested...You will see that in order to write this explanation I had to do quite a number of tests; and although doing a test is quick, doing a lot of tests is slow and boring. So I hope the reader will excuse me if I tell you that other than the two general configuration commands for notes that I mention in the main text and which I focus on in the following explanation, I will leave out another four potential configuration possibilities in the explanation:

- \setupnotes and \setupnotations: In other words, the same name but in the plural. The wiki says that the singular and plural versions of the command are synonymous, and I believe it.

- \setupfootnotes and \setupendnotes: We assume these are specific applications for, foot-

`\setupnotation[NoteType][Configuration]`

where *NoteType* refers to the kind of note we are configuring (`footnote`, `endnote` or the name of some note type we ourselves have created), and *configuration* contains the particular configuration options for the command.

The problem is that the names of these two commands do not make it very clear what the difference is between them or what things each configures; and the fact that many of the options for these commands are not documented does not help much either. After a lot of testing I haven't been able to reach any conclusion that would allow me to understand why certain things are configured with one, while others are configured with the other,[1] except perhaps that, because of the choices I have made to make it work, `\setupnotation` always affects the note text, or the ID that is printed with the note text, while `\setupnote` has some options that affect the mark for the note inserted in the main text.

I will now try to organise what I have found out after doing some tests with the different options of both commands. I leave most of the options for both aside, as they are not documented and I have not been able to draw any conclusions as to what they are for or under what conditions they should be used:

- **ID used for the mark:**

  Notes are always identified by a number. What we can configure here is:

  – *The first number*: controlled by `start` in `\setupnotation`. Its value has to be a whole number, and it uses this to begin counting notes.

  – *The numbering system*, which depends on the `numberconversion` option in `\setupnotation`. Its values can be:

notes and endnotes respectively. Perhaps explaining note configuration on the basis of these commands would be easier, however, since I couldn't get the first option (`numberconversion`) which I tried with `\setupfootnotes` to work, although I know that other options of these commands do work... I was too lazy to add the tests needed to include these two commands in the explanation to the many tests I already had to do to write what follows.

But I am of the opinion (from the few random tests I did) that everything that works in these two commands, but whose explanation I am leaving out, also works in the commands for which I do give an explanation.

[1] There is a page in the ConTEXt wiki that I discovered by chance (since it is not specifically dedicated to notes), which suggests that the difference is that `\setupnotation` controls the text of the note to be inserted, and `\setupnote` the environment of the note in which it will be placed (?) But this is inconsistent with the fact that, for example, the width of the note text (which has to do with its *insertion*) is controlled by the `width` option of `\setupnote` and not by the `\setupnotation` option with the same name. What is controlled here is the width of the space between the mark and the note text.

  - ⋆ *Arabic numerals*: `n, N` or `numbers`.
  - ⋆ *Roman numerals*: `I, R, Romannumerals, i, r, romannumerals`. The first three are upper case Roman numerals and the last three lower case.
  - ⋆ *Numbering with letters*: `A, Character, Characters, a, character, characters` depending on whether we want the letters to be in upper case (the first three options) or lower case (the rest).
  - ⋆ *Numbering with words*. In other words, we write the word that designates the number and so, for example, '3' becomes 'three'. Two methods are possible. The `Words` option writes the words in upper case and `words` in lower case.
  - ⋆ *Numbering with symbols*: we can use four different sets of symbols depending on the option chosen: `set 0, set 1, set 2` o `set 3`. On page 137 there is an example of the symbols used in each of these options.

- *The event that determines restarting note numbering*: This depends on the `way` option in `\setupnotation`. When the value is `bytext` all notes in the document will be numbered sequentially without the numbering being reset. When it is `bychapter, bysection, bysubsection, etc.`, the note counter will be reset each time the chapter, section or subsection is changed, while when it is `byblock` it resets the numbering each time we change blocks in the document macrostructure (see section 7.6). The `bypage` value causes the note counter to restart each time the page is changed.

- **Configuring the note mark:**
  - Whether or not to show it: `number` option in `\setupnotation`.
  - Placement of the mark in relation to the note text: The `alternative` option in `\setupnotation`: it can take any of the following values: `left, inleft, leftmargin, right, inright, rightmargin, inmargin, margin, innermargin, outermargin, serried, hanging, top, command`.
  - Format of the mark in the note itself: The `numbercommand` option in `\setupnotation`.
  - Format of the mark in the body of the text: The `textcommand` option in `\setupnote`.

    The `numbercommand` and `textcommand` options must consist of a command that takes the contents of the mark as an argument. It can be a self-defined command. However,

I have found that simple formatting commands (\bf, \it, etc.) work, although they are not commands that need to take an argument.

- Distance between the mark and the text (in the note itself): The `distance` and `width` options in \setupnotation. I was unable to discover the difference (if indeed there is one) between using one or the other option.
- Existence or not of a hyperlink allowing to jump between the mark in the main text and the mark in the note itself: The `interaction` option in \setupnote. With `yes` as a value there will be a link, and with `no` there will not be one.

- **Configuring the note text itself.**

  We can influence the following aspects:

  - Placement: this depends on the `location` option in \setupnote.

    In principle we already know that footnotes are placed at the bottom of the page (`location=page`) and endnotes at the point at which the \placenotes[endnote] (`location=text`) command is found, however we can adjust this function and set footnotes, for example, as `location=text` which will cause footnotes to work similarly to endnotes so they appear at the point in the document where the \placenotes[footnote] command is found, or the command specific to footnotes \placefootnotes. With this procedure we could, for example, print the notes under the paragraph in which they are found.

  - Paragraph separation between notes: by default each note is printed in its own paragraph, but we can have all the notes on the same page printed in the same paragraph by setting the `paragraph` option in \setupnote to "`yes`".
  - Style in which the note text itself will be written: the `style` option in \setupnotation.
  - Letter size: the `bodyfont` option in \setupnote.

    This option is only for the case where we want to manually set a font size for the footnotes. It is almost never a good idea to do this as, by default, ConTEXt adjusts the font size of the footnotes so that it is smaller than the main text, but with a size *that is proportionate* to that of the font size in the main body.

  - Left margin for the note text: the `margin` option in \setupnotation.
  - Maximum width: the `width` option in \setupnote.
  - Number of columns: the `n` option in \setupnote determines that the note text will be in two or more columns. The 'n' value has to be a whole number.

- **Space between notes or between notes and text:** here, we have the following options:

- `rule`, in `\setupnote` establishes whether or not there will be a line (rule) between the note area and the area of the page with the main text. Its possible values are `yes, on, no` and `off`. The first two enable the rule and the last disables it.
- `before`, in `\setupnotation`: command or commands to be run before inserting the note text. Serves to insert additional spacing, dividing lines between notes, etc.
- `after`, in `\setupnotation`: command or commands to be run after inserting the note text.

12.1.6 Temporary excluding notes when compiling

The `\notesenabledfalse` and `\notesenabledtrue` commands tell ConTEXt to enable or disable compiling of notes respectively. This function can be useful if we wish to obtain a version without notes when the document has numerous and extensive notes. In my personal experience, for example, when I am correcting a doctoral thesis, I prefer to read it the first time in one go, without the notes, and then do a second reading with the notes incorporated.

12.2 Paragraphs with multiple columns

Typesetting the text in more than one column is a possibility that can be established:

a. As a general feature of the page layout.

b. As a feature of certain constructions such as, for example, structured lists, or footnotes or endnotes.

c. As a feature applied to particular paragraphs in a document.

In any of these cases, most of the commands and environments will work perfectly even if we are working with more than one column. There are however some limitations; mainly in relationship to floating objects in general (see section 13.1) and with tables in particular (section 13.3) even if they are not floats.

With regard to the number of columns allowed, ConTEXt has no theoretical limit. However, there are physical limits that have to be taken into account:

- The width of the paper: an unlimited number of columns requires an unlimited width of paper (if the document is to be printed) or screen (if it is a document intended to be displayed on screen). In practice, taking into account the *normal*

width of the paper sizes that are marketed and used to make up books, and the screens of computer devices, it is difficult for a text made up of more than four or five columns to fit well.

- The size of the computer memory: the ConTEXt reference manual points out that, in *normal* systems (neither particularly powerful nor particularly limited in resources), between 20 and 40 columns can be handled.

In this section I will focus on the use of the multi-column mechanism in special paragraphs or fragments, since

- Multiple columns as a page layout option have already been discussed (in subsection B of section 5.3.4).
- The possibility offered by certain constructions, such as structured lists or footnotes, typesetting text in more than one column, is discussed in relation to the construct or environment in question.

12.2.1 The `\startcolumns` environment

The normal procedure for inserting fragments made up of several columns into a document is to use the `columns` environment whose format is:

```
\startcolumns[Configuration] ... \stopcolumns
```

where *Configuration* allows us to control many aspects of the environment. We can indicate the desired configuration each time we call the environment, or adapt the default operation of the environment for all calls to the environment, the latter to be achieved with

```
\setupcolumns[Configuration]
```

In both cases the configuration options are the same. The most important ones, ordered according to their function, are the following:

- **Options that control the number of columns and the space between them:**
  - `n`: controls the number of columns. If this is omitted, two columns will be generated.
  - `nleft, nright`: these options are used in two-sided document layout (see subsection A of section 5.3.4), to establish the number of columns on left (even) and right (odd) pages respectively.
  - `distance`: space between columns.
  - `separator`: determines what marks the separation between columns. It can be `space` (default value) or `rule` in which case a line (rule) will be gene-

rated between the columns. In the event that a rule is established between columns, this rule can in turn be configured with the following two options:

  - ⋆ `rulecolor`: colour of the line.
  - ⋆ `rulethickness`: thickness of the line.

- `maxwidth`: maximum width that columns can have + the space between them.

- **Options that control text distribution in columns:**
  - `balance`: by default, ConTEXt *balances* columns, meaning it distributes the text between them so that they have more or less the same amount of text. However, we can set this option with the "`no`" the text will not start in a column until the previous one is full.
  - `direction`: determines in which direction the text is distributed between the columns. By default, the natural reading order is followed (from left to right), but giving this option the `reverse` value results in right to left.

- **Options affecting typesetting of text within the environment:**
  - `tolerance`: text written in more than one column means that line width within a column is smaller, and as explained when describing the mechanism ConTEXt uses for constructing lines (section 11.3), this makes it difficult to locate optimal points for inserting line breaks. This option allows us to temporarily alter the horizontal tolerance in an environment (see section 11.3.3), to facilitate the typesetting of the text.
  - `align`: controls the horizontal alignment of lines within the environment. It can take any of the following values: `right, flushright, left, flushleft, inner, flushinner, outer, flushouter, middle` or `broad`. Regarding the significance of these options see section 11.6.1.
  - `color`: specifies the name of the colour in which the text within the environment will be written.

12.2.2 Parallel paragraphs

A specific version of the multi-column composition is parallel paragraphs. In this type of construction the text is distributed across two columns (usually, although sometimes more than two), but it is not allowed to flow freely between them, and instead maintains strict control over what will appear in each column. This is very useful, for example, to generate documents which contrast two versions of a text, such as the new and the old version of a recently amended law, or in bilingual

editions; or also to write glossaries for specific text definitions where the text to be defined appears on the left and the definition on the right, etc.

We would normally use the table mechanism to process these kinds of paragraphs; but this is because most text processors are not as powerful as ConTEXt which has the `\defineparagraphs` and `\setupparagraphs` commands that build this type of paragraph using the column mechanism, which, although it has limitations, is more flexible than the table mechanism.

> As far as I know these paragraphs have no special name. I have called them "parallel paragraphs" because to me it seems to be a more descriptive term than the one ConTEXt uses to refer to them: "*paragraphs*".

The basic command here is `\defineparagraphs` whose syntax is:

```
\defineparagraphs[Name][Configuration]
```

where *Name* is the name we give this construction, and *Configuration* are the features it will have, which can also be set later with

```
\setupparagraphs[Name][Column][Configuration]
```

where *Name* is the name given when creating it, *Column* is an optional argument allowing us to configure a particular column, and *Configuration* allows us to determine how it works in practice.

For example:

```
\defineparagraphs
  [MurciaFacts]
  [n=3, before={\blank},after={\blank}]

\setupparagraphs
  [MurciaFacts][1]
  [width=.1\textwidth, style=bold]

\setupparagraphs
  [MurciaFacts][2]
  [width=.4\textwidth]
```

The above fragment would create a three-column environment called MurciaFacts and then set the first column to take up 10 percent of the line width and be written in bold, and set the second column to take up 40 percent of the line width. As the third column is not configured, it will have the remaining width, i.e. 50%.

Once the environment has been created, we can use it to write a brief history of Murcia:

```
\startMurciaFacts
  825
\MurciaFacts
  City of Murcia founded.
\MurciaFacts
  The origins of the city of Murcia are uncertain, but there is evidence
  that it was ordered to be founded under the name of Madina (or Medina)
  Mursiya in the year 825 by the Emir of al-Àndalus Abderramán II,
  probably built over a much earlier settlement.
\stopMurciaFacts
```

825 City of Murcia founded.

The origins of the city of Murcia are uncertain, but there is evidence that it was ordered to be founded under the name of Madina (or Medina) Mursiya in the year 825 by the Emir of al-Àndalus Abderramán II, probably built over a much earlier settlement.

If we wanted to continue telling the story of Murcia, a new instance of the environment (`\startMurciaFacts`) would be needed for the next event, because it is not possible to include several *rows* with this mechanism.

From the example just given, I would like to highlight the following details:

- Once the environment has been created with, say, `\defineparagraphs[MaryPoppins]`, this becomes a normal environment which starts with `\startMaryPoppins` and ends with `\stopMaryPoppins`.
- A `\MaryPoppins` command is also created, used within the environment to indicate when to change the column.

As for the configuration options for parallel paragraphs (`\setupparagraphs`), I understand that, at this stage of the introduction, and taking into account the example just given, the reader is already prepared to work out the purpose of each of the options, so below I will only indicate the name and type of the options and, where appropriate, the possible values. Remember, though, that `\setupparagraphs [Name] [Configuration]` sets up configurations that affect the whole environment, while `\setupparagraphs [Name] [NumColumn] [Configuration]` applies configurations exclusively to the column indicated.

- `n`: Number
- `before`: Command
- `after`: Command
- `width`: Dimension
- `distance`: Dimension
- `align`: Derived from `\setupalign`
- `inner`: Command
- `rule`: on off
- `rulethickness`: Dimension
- `rulecolor`: Rule colour
- `style`: Style Command
- `color`: Colour

The above list of options is not complete; I have excluded from the list of options those that

I would not normally explain here. I have also taken advantage of the fact that we are in the section dedicated to columns to show the list of options in triple columns, although I have not done it with any of the commands explained in this section, but with the `columns` option in the `itemize` environment, to which the next section is dedicated.

12.3 Structured lists

When information is presented in an orderly manner, it is easier for the reader to grasp. But if the arrangement is also visually perceptible, then it highlights for the reader the fact that here we have a structured text. This is why there are certain *constructions* or *mechanisms* that try to highlight the visual arrangement of the text, thus contributing to its structuring. Of the tools that ConTEXt makes available to the author for this purpose, the most important one, which is the subject of this section, is the `itemize` environment that is used to develop what we could call *structured lists*.

The lists consist of a sequence of *text elements* (which I will call *items*), each of them preceded by a character that helps to highlight it by differentiating it from the rest, and which I will call the "separator". The separator can be a number, letter or symbol. Usually (but not always) the *items* are paragraphs, and the list is formatted to ensure the *visibility* of the separator for each element; usually by applying a hanging indent that makes it stand out[1]. In the case of nested lists, the indentation for each is gradually increased. The HTML language usually calls lists where the separator is a number or character that increases sequentially, *ordered lists*, meaning that each *item* of the list will have a different separator that will allow us to refer to each element by its number or identifier; and it gives the name *unordered lists* where the same character or symbol is used for every item in the list.

ConTEXt automatically manages the numbering or alphabetical sequencing of the separator in numbered lists, as well as the indentation that nested lists need to have; and, in the case of nesting unordered lists, it also looks after the selection of a different character or symbol that allows the level of an *item* in the list to be distinguished at a glance according to the symbol that precedes it.

The reference manual says that the maximum level of nesting in lists is 4, but I guess that was the case in 2013, when the manual was written. According to my tests there seems to be no limit to the nesting of *ordered* lists (in my tests I reached up to 15 levels of nesting). While for unordered lists, there does not seem to be a limit either, in the sense that no matter how many nests we include, no error will be generated; but, for unordered lists, ConTEXt only applies default symbols for the first nine levels of nesting.

[1] In typography an indent that applies to all the lines of a paragraph except the first one is called a *hanging indent*, which makes the first word or character of the paragraph easy to find.

> In any case, it should be pointed out that the excessive use of nesting in lists can have the opposite effect to what we intend, and that is that the reader feels lost, unable to locate each item in the general structure of the list. For this reason I personally believe that while lists are a powerful tool for structuring a text, it is almost never a good idea to go beyond the third level of nesting; and even the third level should only be used in certain cases where we can justify it.

The general tool for writing lists in ConTEXt is the `\itemize` environment whose syntax is as follows:

```
\startitemize[Options][Configuration] ... \stopitemize
```

where the two arguments are optional. The first one allows symbolic names as content that have been assigned a precise meaning by ConTEXt; the second argument, which is rarely used, makes it possible to assign specific values to certain variables that affect the functioning of the environment.

12.3.1 Selection the kind of list and separator between *items*

A. Unordered lists

By default the list generated by `itemize` is an unordered list, in which the separator will be automatically selected depending on the nesting level:

- For the first level of nesting.
– For the second level of nesting.
⋆ For the third level of nesting.
▷ For the fourth level of nesting.
◦ For the fifth level of nesting.
○ For the sixth level of nesting.
◯ For the seventh level of nesting.
□ For the eighth level of nesting.
✓ For the ninth level of nesting.

However, we can expressly indicate that we want the symbol associated with a particular level to be used, simply by passing on the level number as an argument. Thus, `\startitemize[4]` will generate an unordered list in which the ▷ character will be used as a separator, regardless of the nesting level of the list.

We can also modify the predetermined symbol for each level with `\definesymbol`:

```
\definesymbol[Level]{Symbol associated with the level}
```

For example

```
\definesymbol[1][\diamond]
```

will cause the first level of unordered lists to use a ⋄ symbol. With this same command we can assign some symbols to nesting levels higher than nine. Thus, for example

```
\definesymbol[10][\copyright]
```

will assign the international *copyright* symbol: © to nesting level 10.

B. Ordered lists

To generate an ordered list we need to tell `itemize` the kind of ordering we want. It can be:

n	1, 2, 3, 4, ...	**a**	a, b, c, d, ...	**r**	i, ii, iii, iv, ...
m	1, 2, 3, 4, ...	**A**	A, B, C, D, ...	**R**	I, II, III, IV, ...
g	α, β, γ, δ, ...	**KA**	A, B, C, D, ...	**KR**	I, II, III, IV, ...
G	A, B, Γ, Δ, ...				

The difference between `n` and `m` lies in the font used to represent the number: `n` uses the font enabled at that moment, while `m` uses a different, more elegant, almost calligraphic font.

> I do not know the name of the font that `m` uses, and therefore in the above list I have not been able to represent exactly the type of numbers that this option generates. I suggest readers test it out for themselves.

12.3.2 Inputting the items in a list

As a rule, the items in a list created with `\startitemize` are input with the `\item` command that also has a version in environment form that is more suited to the Mark IV style: `\startitem ... \stopitem`. Thus the following example:

```
\startitemize[a, packed]
\startitem First element \stopitem
\startitem Second element \stopitem
\startitem Third element \stopitem
\stopitemize
```

a. First element
b. Second element
c. Third element

produces exactly the same result as

```
\startitemize[a, packed]
\item First element
\item Second element
\item Third element
\stopitemize
```

a. First element
b. Second element
c. Third element

`\item` or `\startitem` is the *general* command for introducing an item into the list. Along with it there are the following additional commands for when we want to achieve a special result:

`\head` This command should be used in place of `\item` when we want to avoid inserting a page break after the item in question.

> A common construction is to include a nested list or a text block immediately below a list element, so that the list element, in a sense, functions as the *title* of the sub-list or text block. In these cases a page break between that element and the subsequent paragraphs would be inadvisable. If we use `\head` instead of `\item` to input these elements ConTEXt *will endeavour* (as far as possible) not to separate such element from the next block.

`\sym` The `\sym{Text}` command inputs an item in which the text used as an argument of `\sym` is used as a *separator*, not a number or symbol. This list, for example, is constructed with items input by means of `\sym` instead of `\item`.

`\sub` This command, which should be used only in ordered lists (where each item is preceded by a number or letter in alphabetical sequence), causes the item input with it to keep the number of the previous item, and in order to indicate that the number is repeated and that this is not a mistake, the '+' sign is printed on the left. This can be useful if we are referring to a previous list for which we suggest modifications but where, for the sake of clarity, the numbering of the original list should be maintained.

`\mar` This command maintains the numbering of the items, but adds a letter or character in the margin (which is passed to it as an argument, between curly brackets). I'm not quite sure how useful it is.

There are two additional commands for inputting items, whose combination produces very *interesting* effects and, if I may say so, I think it is better to explain them with an example. `\ran` (abbreviation of *range*) and `\its`, abbreviation of *items*. The first one takes an argument (between curly brackets) and the second repeats the symbol used as a separator in the list x number of times (by default 4 times, but we can alter that by using the `items` option). The following example shows how these two commands can work together to create a list that mimics a questionnaire:

```
After reading the following introduction, answer the following questions:

\startitemize[5, packed][width=8em, distance=2em, items=5]
\ran{No \hss Yes}
\its I will never use \ConTeXt, it is too difficult.
\its I will only use it for writing big books.
\its I will always use it.
\its I like it so much I will call my next child \quotation{Hans}, as a tribute
to Hans Hagen.
\stopitemize
```

After reading this introduction, answer the following questions:

No				Yes	
∘	∘	∘	∘	∘	I will never use ConTEXt, it is too difficult.
∘	∘	∘	∘	∘	I will only use it for writing big books.
∘	∘	∘	∘	∘	I will always use it.
∘	∘	∘	∘	∘	I like it so much I will call my next child “Hans”, as a tribute to Hans Hagen.

12.3.3 Basic list configuration

We recall that “`itemize`” allows for two arguments. We have already seen how the first argument lets us select the type of list we want. But we can also use it to indicate other characteristics of the list; this is done through the following options for “`itemize`” in its first argument:

- `columns`: this option determines that the list is made up of two or more columns. After the columns option, the desired number of columns must be written as words separated by a comma: two, three, four, five, six, seven, eight or nine. `Columns` not followed by any number generates two columns.
- `intro`: this option tries not to separate the list, by a line break, from the paragraph that precedes it.
- `continue`: in ordered lists (numerical or alphabetical) this option causes the list to continue the numbering from the last numbered list. If the `continue` option is used, it is not necessary to indicate what type of list we want, as it is assumed that it will be the same as the last numbered list.
- `packed`: is one of the most used options. Its use causes the vertical space between the different *items* on the list to be reduced as far as possible.
- `nowhite`: produces an effect similar to `packed`, but more drastic: not only does it reduce the vertical space between the items, but also the vertical space between the list and the surrounding text.
- `broad`: increases the horizontal space between the item separator and the item text. The space can be increased by multiplying a number by `broad` as in, for example `\startitemize[2*broad]`.
- `serried`: removes the horizontal space between the item separator and the text.
- `intext`: removes the hanging indent.
- `text`: removes the hanging indent and reduces vertical space between items.

- `repeat`: in nested lists makes the numbering of a child level *repeat* the same level as the previous level. This way we would have, on the first level: 1, 2, 3, 4; on the second level: 1.1, 1.2, 1.3, etc. The option must be indicated for the inside list, not on the outer list.

- `margin, inmargin`: by default the list separator is printed on the left, but within the text area itself (`atmargin`). The options `margin` and `inmargin` move the separator to the margin.

12.3.4 Additional list configuration

The second argument, also optional, in `\startitemize` allows for a more detailed and thorough configuration of lists.

- `before, after`: commands to be run before starting or after closing, the itemize environment, respectively.

- `inbetween`: command to be run between two `items`.

- `beforehead, afterhead`: command to be run before or after an item input with the `\head` command.

- `left, right`: character to be printed to the left or right of the separator. For example, to get alphabetical lists in which the letters are surrounded by parentheses we would have to write:

  ```
  \startitemize[a][left=(, right=)]
  ```

- `stopper`: indicates a character to be written after the separator. Only works in ordered lists.

- `width, maxwidth`: width of the space reserved for the separator and, therefore, for the hanging indent.

- `factor`: representative number of the separation factor between the separator and the text.

- `distance`: measure of the distance between the separator and the text.

- `leftmargin, rightmargin, margin`: margin to be added to the left (leftmargin) or right (rightmargin) of the items.

- `start`: number from which the numbering of items will start.

- `symalign, itemalign, align`: alignment of items. Allows for the same values as `\setupalign`. `symalign` controls alignment of the separator; `itemalign` the item text, and `align` alignment of both.

- `identing`: indentation of the first line in the paragraphs within the environment. See section 11.1.1
- `indentnext`: indicates whether the paragraph after the environment should be indented or not. Values are *yes*, *no* and `auto`.
- `items`: in items entered input with `\its`, indicates the number of times the separator must be reproduced.
- `style, color; headstyle, headcolor; marstyle, marcolor; symstyle, symcolor`: these options control the style and colour of the items as they are input into the environment with `\item`, `\head`, `\mar` or `\sym` commands.

12.3.5 Simple lists with the `\items` command

An alternative to the `itemize` environment for very simple unnumbered lists, where the items are not too big is the `\items` command whose syntax is:

```
\items[Configuration]{Item 1, Item 2, ..., item n}
```

The different items that the list will have are separated from each other by commas. For example:

```
Graphics files can
have, among other things, the
following extensions:

\items{png, jpg, tiff, bmp}
```

Graphics files can have, among other things, the following extensions:

- png
- jpg
- tiff
- bmp

The configuration options supported by this command are a subset of the `itemize` ones, except for two specific options for this command:

- `symbol`: this option determines the type of list that will be generated. It supports the same values used for `itemize` to select some type of list.
- `n`: this option indicates from which item number there will be a separator.

12.3.6 Predetermining list behaviour and creating our own list types

In the previous sections we have seen how to indicate what type of list we want and what characteristics it should have. But doing that every time a list is called is

inefficient and will usually produce an incoherent document in which each list has its own appearance, but without the different appearances meeting any criteria.

Preferable result for this:

- Predetermine the *normal* behaviour of `itemize` and `\items` in the document's preamble.
- Create our own customised lists. For example: an alphabetically numbered list we want to call *ListAlpha*, a list numbered with Roman numerals (*ListRoman*), etc.

We achieve the first with the `\setupitemize` and `\setupitems` commands. The second requires the use of either the `\defineitemgroup`, or `\defineitems` command. The first will create a list environment similar to `itemize` and the second a command similar to `items`.

12.4 Descriptions and enumerations

Descriptions and enumerations are two constructions that allow for the consistent typesetting of paragraphs or groups of paragraphs that develop, describe, or define a phrase or word.

12.4.1 Descriptions

For descriptions we differentiate between a *title* and its explanation or development. We can create a new description with:

```
\definedescription[Name] [Configuration]
```

where *Name* is the name by which this new construction will be known, and Configuration controls what our new structure will look like. After the previous statement we will have a new command and an environment with the name we have chosen. Thus:

```
\definedescription[Concept]
```

will create the commands:

```
\Concept{Title}
\startConcept {Title} ... \stopConcept
```

We will use the command for the case where the explanatory text of the title consists of only one paragraph, and the environment for titles whose description occupies more than one paragraph. When the command is used, the paragraph immediately following it is the one that will be considered the title's explanatory

text. But when the environment is used, all content will be formatted with the appropriate indentation to make it clear how it relates to the title.

For example:

```
\definedescription
  [Concept]
  [alternative=left, width=1cm, headstyle=bold]

\Concept{Contextualise}

Place something in a certain context, or typeset a text with the typesetting
system called \ConTeXt. The ability to correctly contextualise in any situation
is considered a sign of intelligence and good sense.
```

This will generate the following result:

Contextualise Place something in a certain context, or typeset a text with the typesetting system called ConTEXt. The ability to correctly contextualise in any situation is considered a sign of intelligence and good sense.

As is normally the case with ConTEXt, the appearance that our new construction will have can be indicated at the time of its creation, with the *Configuration* argument or later on with `\setupdescription`:

```
\setupdescription[Name] [Configuration]
```

where *Name* is the name of our new description, and *Configuration* determines what it looks like. Among the different possible configuration options I will highlight:

- `alternative`: This option is the one that fundamentally controls the appearance of the construction. It determines the placement of the title in relation to its description. Its possible values are `left, right, inmargin, inleft, inright, margin, leftmargin, rightmargin, innermargin, outermargin, serried, hanging`, their names are clear enough to get an idea of the result, although, in case of doubt, it is best to do a test to see how it looks.

- `width`: controls the width of the box in which the title will be written. Depending on the value of `alternative` that distance will also be part of the indentation with which the explanatory text is written.

- `distance`: controls the distance between the title and the explanation.

- `headstyle, headcolor, headcommand`: affect how the title itself will look: Style (`headstyle`) and color (`headcolor`). With headcommand we can indicate

a command to which the title text will be passed as an argument. For example: `headcommand=\WORD` will make sure that the title text is all in upper case.

- `style, color`: controls the appearance of the title's descriptive text.

12.4.2 Enumerations

Enumerations are numbered text elements structured on several levels. Each element starts with a title that consists, by default, of the name of the structure and its number, although we can change the title with the `text` option. They are quite similar to descriptions, although they differ in that:

- All the elements in an enumeration share the same title.
- Therefore they differ from each other by their numbering.

This structure can be very useful, for example, to write formulas, problems or exercises in a textbook, ensuring that they are numbered correctly and formatted in a consistent manner.

We create an enumeration with

```
\defineenumeration[Name] [Configuration]
```

where *Name* is the name of the new construction, and *Configuration* controls what it will look like.

So, in the following example:

```
\defineenumeration
  [Exercise]
  [alternative=top, before=\blank, after=\blank, between=\blank]
```

We have created a new structure called *Exercise* and, as happens with enumerations, we will have the following new commands available:

```
\Exercise
\startExercise
```

The command is used just for a single paragraph *exercise*, and the environment for multiple paragraph *exercises*. But since enumerations can be up to four levels deep, the following commands and environments will also be created:

```
\subExercise
\startsubExercise
\stopsubExercise
\subsubExercise
\startsubsubExercise
\stopsubsubExercise
```

```
\subsubsubExercise
\startsubsubsubExercise
\stopsubsubsubExercise
```

And, to control the numbering, the following additional commands:

- `\setEnumerationName`: sets the current numbering value.
- `\resetEnumerationame`: sets the enumeration counter to zero.
- `\nextEnumerationName`: increases the enumeration counter by one.

The appearance of enumerations can be determined at the time of their creation or later with `\setupenumeration` whose format is:

`\setupenumeration[Name] [Configuration]`.

For each enumeration we can configure each of its levels separately. Thus, for example, `\setupenumeration [subExercise] [Configuration]` will affect the second level of the enumeration called “Exercise”.

The options and values configurable with `\setupenumeration` are similar to those in `\setupdescription`.

12.5 Lines and frames

It says in theConTEXt reference manual that TEX has a huge text management capability, but is very weak in managing graphic information. I beg to differ: it is true that for handling lines and frames the possibilities of ConTEXt (actually TEX) are not as overwhelming as when it comes to typesetting text. But to go on to say that the system is weak in this regard is, I think, somewhat of a stretch. I don't know of any function with lines and frames that other typesetting systems can do for documents that ConTEXt is unable to generate. And if we combine ConTEXt with MetaPost, or with TiKZ (ConTEXt has an expansion module for this), then the possibilities are only limited by our imagination.

In the following sections, however, I will limit myself to explaining how to generate simple horizontal and vertical lines and frames around words, sentences or paragraphs.

12.5.1 Simple lines

The simplest way of drawing a horizontal line is with the `\hairline` command that generates a horizontal line that occupies all the width of a normal text line.

There cannot be text of any kind on the line where the line generated by \hairline is. In order to generate a line capable of coexisting with the text on the same line, we need the \thinrule command. This second command will use the full width of the line. Therefore, in an isolated paragraph, it will have the same effect as \hairline, while in the opposite case, \thinrule will produce the same horizontal expansion as \hfill (see section 10.3.3), but instead of filling the horizontal space with white space (as \hfill does), it fills it with a line.

```
On the left\thinrule\\
\thinrule On the right\\
On both\thinrule sides\\
\thinrule centred\thinrule
```

On the left
On the right
On both sides
centred

With the \thinrules command we can generate several lines. For example \thinrules[n=2] will generate two consecutive lines, each the width of the normal line. The lines generated with \thinrules can also be configured, either in an actual call to the command, indicating the configuration as one of its arguments, or generally with \setupthinrules. Configuration includes the thickness of the line (rulethickness), its colour (color), background colour (background), interline space (interlinespace), etc.

> I will leave a number of options without explanation. The reader can consult them in setup-en.pdf (see section 3.6). Some options only differ from others in terms of nuance (i.e there is hardly any difference between them), and I think it is faster for the reader to try to *see* the difference, than for me to try to convey it in words. For example: the thickness of the line I just said depends on the rulethickness option. But it is also affected by the height and depth options.

Smaller lines can be generated with the \hl and \vl commands. The first generates a horizontal line and the second a vertical line. Both take a number as a parameter that allows us to calculate the length of the line. In \hl the number measures the length in *ems* (no need to indicate the unit of measurement in the command) and in \vl the argument refers to the current height of the line.

Thus \hl[3] generates a horizontal line of 3 ems and \vl[3] generates a vertical line of the height corresponding to three lines. Remember that the line measurement indicator must be inserted between square brackets, not between curly brackets. In both commands the argument is optional. If it is not entered, a value of 1 is assumed.

\fillinline is another command to create horizontal lines of precise length. It supports more configuration in which we can indicate (or predetermine with \setupfillinlines) the width (width option) in addition to some other features.

A peculiarity of this command is that text that is written to its right will be placed on the left of the line, separating that text from the line by the necessary white space to occupy the whole line. For example:

```
\fillinline[width=6cm] Name
```

will generate

Name

I suspect that this strange operation is due to the fact that this macro was designed to write forms where there is a horizontal line behind the text on which something must be written. In fact the very name of the command `fillinline` means, fill in the line.

Besides the width of the line, we can configure the margin (`margin`), the distance (`distance`), the thickness (`rulethickness`) and the colour (`color`).

Almost identical to `\fillinline` is `\fillinrules`, although this command allows us to insert more than one line ("`n`" option).

```
\fillinrules[Configuration] {Text} {Text}
```

where the three arguments are optional.

12.5.2 Lines linked to text

Although some of the commands we have just seen can generate lines that coexist with text on the same line, those commands actually focus on the line's layout. To write lines linked to certain text, ConTEXt has commands:

- that generate text between lines.
- that generate lines under the text (underlining), above the text (overlining) or through it (strikethrough).

To generate a text between lines the usual command is `\textrule`. This command draws a line that crosses the entire width of the page and writes the text it takes as a parameter on the left side (but not at the margin). For example:

```
\textrule{Example text}
```

Example text

It is assumed that `\textrule` allows an optional first argument with three possible values: `top`, `middle` and `bottom`. But, after some tests, I have not been able to find out what effect such options produce.

Similar to \texrule is the \starttextrule environment which, besides inserting the line with text at the beginning of the environment, inserts a horizontal line at the end. The format of this command is:

```
\starttextrule[Configuration]{Text on the line} ... \stoptextrule
```

Both \textrule and starttextrule can be configured with \setuptextrule.

To draw lines under, over, or through text, the following commands are used:

```
\underbar{Text}
\underbars{Text}
\overbar{Text}
\overbars{Text}
\overstrike{Text}
\overstrikes{Text}
```

As we can see, for each type of line (under, over, or through text) there are two commands. The singular version of the command draws a single line under, over or through all the text taken as an argument, while the plural version of the command only draws the line over the words, but not the white space.

These commands are not compatible with each other, that is to say, two of them cannot be applied to the same text. If we try, the last one will always prevail. On the other hand \underbar can be nested, underlining what has already been underlined.

> The reference manual points out that \underbar disables hyphenation of words in the text that constitute its argument. It is not clear to me whether that statement refers only to \underbar or to the six commands we are examining.

12.5.3 Framed words or texts

To surround a text with a frame or grid we use:

- The \framed or \inframed commands if the text is relatively brief and does not take up more than one line.
- The \startframedtext environment for longer texts.

The difference between \framed and \inframed lies at the point from which the frame is drawn. In \frame the frame is drawn upwards from an ideal line, called a baseline, on which the letters rest, but certain letters pass downwards. In \inframed the frame is drawn, also upwards, from the lowest possible point on the line. For example:

```
Here there are \framed{two} good
\inframed{frames}.
```

Here there are two good frames.

Both, framed and inframed text, can be customized with `\setupframed`, and `\startframedtext` is customized with `\setupframedtext`. The customization option for both commands are quite similar. They allow us to indicate the measurements of the frame (`height, width, depth`), the shape of the corners (`framecorner`), which can be `rectangular` or round (`round`), the frame colour (`framecolor`), the line thickness (`framethickness`), the alignment of content (`align`), text colour (`foregroundcolor`), background colour (`background` and `backgroundcolor`), etc.

For `\startframedtext` there is also an apparently strange property: `frame=off` that causes the frame not to be drawn (although it is still there, but invisible). This property exists because since the frame around a paragraph is indivisible, it is common for the entire paragraph to be enclosed in a `framedtext` environment with the frame drawing option turned off, to ensure that no page breaks are inserted within a paragraph.

We can also create a customized version of these commands with `\defineframed` and `\defineframedtext`.

12.6 Other environments and constructions of interest

There are still many environments in ConTEXt that I have not even mentioned, or only very much in passing. By way of example:

- **`buffer`** *Buffers* are text fragments stored in memory for later re-use. A *buffer* is defined somewhere in the document with `\startbuffer[BufferName] ... \stopbuffer` and can be retrieved as often as desired at some other point in the document with `\getbuffer[BufferName]`.

- **`chemical`** This environment allows us to place chemical formulas inside it. If TEX stands out, among many other things, for its ability to typeset texts with mathematical formulas properly, from the outset ConTEXt sought to extend this ability to chemical formulas, and it has this environment where commands and structures are enabled for writing chemical formulas.

- **`combination`** This environment allows us to combine several floating elements on the same page. It is particularly useful for aligning different connected external images in our document.

- **`formula`** This is an environment aimed at typesetting maths formulas.

- **hiding** The text stored in this environment will not be compiled and will not appear, therefore, in the final document. This is useful for temporarily disabling compilation of certain fragments in the source file. The same thing is achieved by marking one or more lines as a comment. But when the fragment we want to disable is relatively long, more effective than marking tens or hundreds of lines of the source file as a comment is to insert the `\starthiding` command at the beginning of the fragment, and `\stophiding` at the end.

- **legend** In a mathematical context, TEX applies different rules so that no normal text can be written. However, sometimes a formula is accompanied by a description of the elements used in it. For this purpose there is the `\startlegend` environment which allows us to place normal text in a mathematical context.

- **linecorrection** Usually, ConTEXt correctly manages the vertical space between lines, but occasionally a line may contain something that makes it not look right. This happens mainly with lines that have fragments framed with `\framed`. In such cases this environment adjusts the line spacing so that the paragraph appears correctly.

- **mode** This environment is intended to include fragments in the source file that will only be compiled if the appropriate mode is active. The use of *modes* is not the subject of this introduction, but it is a very interesting tool if we want to be able to generate several versions with different formats, from a single source file. A complementary environment to this one is `\startnotmode`.

- **opposite** This environment is used to typeset texts when the contents of the left and right pages are related.

- **quotation** A very similar environment to **narrower**, intended to insert moderately long literal quotations. The environment makes sure that the text inside is quoted, and that the margins are increased so that the paragraph with the quotation stands out visually on the page. But it should be noted that according to usual blockquote style in English, there should be no opening and closing quotation marks – which makes this command or environment less useful.

- **standardmakeup** This environment is designed to generate pages with the title of the document, which is relatively common in academic documents of a certain length, such as doctoral theses, master's theses, etc.

To learn about any of these environments (or others I have not mentioned), I suggest the following steps:

1. Look for information on the environment in the ConTEXt reference manual. This manual does not mention all the environments; but it does say something about every item in the list above.

2. Write a test document where the environment is used.

3. Look up ConTEXt's official list of commands (see section 3.6) for the configuration options for the environment in question, then test them to see exactly what they do.

Chapter 13
Images, tables and other floating objects

This chapter is mainly about floating objects (floats). But following up on this concept, it takes advantage of it to explain two object types that are not necessarily floats, although they are often configured as if they were: external images and tables. Looking at this chapter's table of contents, one might think this is all very untidy: it begins by talking about floating objects, then goes on to talk about images and tables, and finishes by once again talking about floating objects. The reasons for this untidiness are *pedagogical*: images and tables can be explained without insisting too much on the fact that they are normally floats; and yet, when we start examining them it helps a lot to discover that, surprise surprise, we already know about two floating objects.

13.1 What are floating objects and what do they do?

If a document were to contain only *normal* text, paginating it would be relatively easy: knowing the maximum height of the text area of the page is enough to measure the height of the different paragraphs to know where to insert page breaks. The problem is that in many documents there are objects, fragments or indivisible blocks of text such as an image, a table, a formula, a framed paragraph, etc.

Sometimes these objects can occupy a large portion of the page, which in turn poses the problem that if you have to insert it at a particular point in the document, it may not fit on the current page, and has to be interrupted abruptly, leaving a large blank space at the bottom, so that the object in question, and the text that follows it, are moved to the next page. The rules of good typesetting, however, indicate that, except for the last page of a chapter, there should be the same amount of text on each page.

It is therefore advisable to avoid large blank vertical spaces appearing; and *floating* objects are the main mechanism for achieving this. A "floating object" is one that does not have to be located at an exact point in the document, but can *move* or *float* around it. The idea is to allow ConTEXt to decide on the best place, from a pagination point of view, to locate such objects, even authorising them to move to another page; but always trying not to move too far away from the point of inclusion in the source file.

Therefore, there are no objects that have to be floats *per se*. But there are objects that will occasionally need to be floats. The decision, in any case, is up to the author or the person in charge of typesetting, if they are two different people.

Undoubtedly, allowing the exact placement of an indivisible object to change, very much facilitates the task of typesetting nicely balanced pages; but the problem that goes with this is that since we don't know exactly where such an object will end up at the time we are writing the original, it is difficult to make reference to it. So, for example, if I have just put a command in my document that inserts an image and in the next paragraph I want to describe it and write something about it like: "As you can see from the previous figure", when the figure *floats* it could well be placed *after* what I have just written and the result is an inconsistency: the reader is looking for an image *before* the text that refers to it and can't find it because after floating, the image has ended up after that reference.

This is fixed by *numbering* floating objects (after distributing them in categories), so that in instead of referring to an image as "the previous image" or "the next image", we will refer to it as "image 1.3", since we can use ConTEXt's internal reference mechanism to ensure that the image number is always kept up to date (see section 9.2). The numbering of these kinds of objects, on the other hand, makes it easier to quite easily create an index of them (index of tables, graphs, images, equations, etc.). For how to do this, see (section 8.2).

The mechanism for dealing with floating objects in ConTEXt is quite sophisticated and so abstract at times that it may not make it suitable for beginners. Therefore, in this chapter I will start by explaining it using two particular cases: images and tables. Then I will try to generalise somewhat so that we can understand how to extend the mechanism to other kinds of objects.

13.2 External images

As the reader at this stage knows (since it has been explained in section 1.5), ConTEXt is perfectly integrated with MetaPost and can generate images and graphics that are *programmed* in much the same way as text transformations are programmed. There is also an extension module for ConTEXt[1] that allows it to work with TiKZ.[2] But such images are not dealt with in this introduction (as this would probably force its length to be doubled). I am referring here to the use of external images, which reside in a file on our hard drive or are downloaded directly from the Internet by ConTEXt.

13.2.1 Directly inserting images

To directly insert an image (not as a floating object) we use the `\externalfigure` command whose syntax is

```
\externalfigure[Name] [Configuration]
```

where

- *Name* can be either the name of the file containing the image, or the web address of an image found on the Internet, or a symbolic name we have previously associated with an image using the `\useexternalfigure` command whose format is similar to that of `\externalfigure` although it takes a first argument with the symbolic name that will be associated with the image in question.
- *Configuration* is an optional argument that allows us to apply certain transformations to the image before it is inserted into our document. We will examine this argument more closely in section 13.2.4.

The image formats allowed are pdf, mps, jpg, png, jp2, jbig, jbig2, jb2, svg, eps, gif or tif. ConTEXt can directly manage eight of these, while the rest (svg, eps, gif or tif) need to be converted with an external tool before opening them, that changes according to the format and therefore must be installed on the system so that ConTEXt can manipulate these kinds of files.

> Among the formats supported by `\externalfigure` are also some video formats. In particular: QuickTime (extension .mov), Flash Video (extension .flv) and MPeg 4 (extension .mp4). But

[1] ConTEXt extension modules give it additional utilities but are not included in this introduction.

[2] This is a graphics programming language intended to work with TEX-based systems. It is a "recursive acronym" from the German sentence "TiKZ ist keinen Zeichenprogramm" which translated means: "TiKZ is not a drawing program". Recursive acronyms are a kind of programmers' joke. Leaving aside MetaPost (which I do not know how to use), I believe that TiKZ is a great system for programming graphics with.

most PDF players do not know how to handle PDF files with video embedded in them. I can't say much about this, as I haven't done any tests.

There is no need to indicate the file extension: ConTEXt will search for a file with the specified name and one of the extensions for the known image formats. If there are several candidates, first the PDF format is used if there is one, and in its absence the MPS format (graphics generated by MetaPost). In the absence of these two, the following order is followed: jpeg, png, jpeg 2000, jbig and jbig2.

If the actual format of the image does not correspond to the extension of the file that stores it, ConTEXt cannot open it unless we indicate the actual format of the image using the `method` option.

If the image is not placed on its own outside of a paragraph, but is integrated into a text paragraph, and its height is greater than the line spacing, the line will be adjusted to prevent the image from overlapping the previous lines, as in the example that accompanies this line  .

By default, ConTEXt searches for the images in the working directory, in its parent directory and in that directory's parent directory. We can indicate the location of a directory containing the images we will work with using the `directory` option of the `\setupexternalfigures` command, which would add that directory to the search path. If we want the search to be performed only in the image directory, we have to set the `location` option as well. So, for example, so that our document looks for all the images we need in the "`img`" directory, we should write:

```
\setupexternalfigures
  [directory=img, location=global]
```

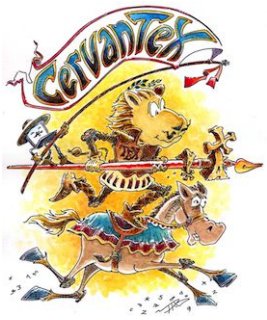
In the `directory` option in `\setupexternalfigures`, we can include more than one directory, separating them with commas; but in this case we need to enclose the directories within curly brackets. For example "`directory={img, ~/imágenes}`".

In `directory` we always use the '/' character as the separator between directories; including in Microsoft Windows whose operating system uses the '\' as its directories separator.

`\externalfigure` is also able to use images hosted on the Internet. So, for example, the following snippet will insert the CervanTeX logo directly from the Internet into the document. This is the TEX Spanish-speaking user group:[1]

[1] Internet addresses are very long, and there is not much space available to display the double-column example. Therefore, in order to make the order in the left column fit properly, I have inserted a line break within the web address. If someone wants to copy and paste the example, it will not work if this line break is not deleted.

```
\externalfigure
[http://www.cervantex.es/files/
cervantex/cervanTeXcolor-small.jpg]
```



When a document containing a remote file is first compiled, it is downloaded from the server and stored in the LuaTeX cache directory. This cached file is used during subsequent compilations. Normally, the remote image is downloaded again if the image in the cache is older than 1 day. To change this threshold see the ConTEXt wiki.

If ConTEXt does not find the image that should be inserted, no error is generated, but instead of the image a text block will be inserted with information about the image that should go there. The size of this block will be the image size (if known by ConTEXt) or, otherwise, a standard size. There is an example of this in section 13.4.3.

13.2.2 Inserting an image with \placefigure

Images can be directly inserted. But it is preferable to do this with `\placefigure`. This command causes ConTEXt:

- to know that an image is being inserted that must be incorporated into the list of images in the document that can then be used, if we wish, to produce an index of images.
- to assign a number to the image, thus facilitating internal references to it.
- to add a title to the image, creating a text block between the image and its title that means these cannot be separated.
- to automatically set the white space (horizontal and vertical) needed for the image to be viewed correctly.
- to position the image in the place indicated, making the text flow around it if necessary.
- to convert the image to a floating object if it is possible, taking into account its size and location specifications.[1]

[1] This latter is my conclusion, given that among the placement options there are ones like `force` or `split` that go against the true notion of the floating object.

The syntax for this command is as follows:

```
\placefigure[Options][Label] {Title} {Image}
```

The various arguments have the following meanings:

- *Options* are a set of indications that generally refer to where to place the image. Since these options are the same in this and other commands, I will explain them together later (in section 13.4.1). For now, I will use the `here` option for examples. It tells ConTEXt that, as far as possible, it should place the image exactly at the point in the document where the command that inserts it is found.
- *Label* is a text string to refer internally to this object so we can make reference to it (see section 9.2).
- *Title* is the title text to be added to the image.
- *Image* is the command that inserts the image.

For example

```
\placefigure
  [here]
  [fig:texknuth]
  {\TeX\ logo and photo of {\sc Knuth}}
  {\externalfigure[https://i.ytimg.com/vi/8c5Rrfabr9w/maxresdefault.jpg]}
```

Figure 13.1 TEX logo and photo of KNUTH

As we can see in the example, by inserting the image (which, by the way, has been done directly from an image hosted on the Internet), there are some changes regarding what happens when using the `\externalfigure` command directly. Vertical space is added, the image is centred and a title added. Those are *external* changes obvious at first sight. From an internal point of view the command has also produced other no less important effects:

- First of all, the image has been inserted into the "list of images" which ConTEXt maintains internally for objects inserted into the document. This, in turn,

means that the image will appear in the image index that can be generated with `\placelist[figure]` (see section 8.2), although there are two specific commands to generate the image index which are `\placelistoffigures` or `\completelistoffigures`.

- Secondly, the image has been linked to the label that was added as the second argument to the `\placefigure` command, which means that from now on we can make internal references to it using that label (see section 9.2).
- Finally, the image has become a float, which means that if, for typesetting needs (pagination) it needed to move, ConTEXt would alter its placement.

Actually, `\placefigure`, despite its name, is not only used for inserting images. We can insert anything with it, including text. However, the text or other items inserted into the document with `\placefigure`, will be treated *as if they were an image*, even though they are not; they will be added to the list of images internally managed by ConTEXt, and we can apply transformations similar to the ones we use for images such as scaling or rotating, framing, etc. Thus the following example:

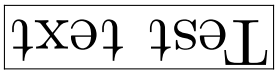


Figure 13.2 Using \placefigure for text transformations

which is achieved as follows:

```
\placefigure
  [here, force]
  [fig:testtext]
  {Using \backslash placefigure for text transformations}
  {\rotate[rotation=180]{\framed{\tfd Test text}}}
```

13.2.3 Inserting images integrated into a text block

Except for very small images, which can be integrated into a line without too much disruption to paragraph spacing, images are usually inserted into a paragraph that contains only them (or put in other words, the image can be thought of as a paragraph in its own right). If the image is inserted with `\placefigure` and its size allows, depending on what we have indicated regarding its placement (see section 13.4.1), ConTEXt will allow the text from the previous and subsequent paragraphs to flow around the image. However, if we want to ensure that a certain image will not be separated from a certain text, we can use the `figuretext` environment whose syntax is as follows:

```
\startfiguretext
  [Options]
  [Label]
```

```
    {Title}
    {Image}

    ... Text

\stopfiguretext
```

The environment's arguments are exactly the same as for `\placefigure` and have the same meaning. But here the options are no longer options for placement of a floating object, but indications regarding the integration of the image into the paragraph; so, for example, "`left`" here means that the image will be placed on the left while text will flow to the right, while "`left, bottom`" will mean that the image must be placed on the lower left side of the text associated with it.

The text written within the environment is what will flow around the image.

13.2.4 Configuration and transformation of images inserted

A. Insert command options that cause some transformation of the image

The final argument in the `\externalfigure` command allows us to carry out certain adjustments to the image inserted. We can make these adjustments:

- In general for all images to be inserted in the document; or for all images to be inserted from a certain point. In this case we make the adjustment with the `\setupexternalfigures` command.
- For a specific image that we want to insert several times in the document. In this case the adjustment is made in the last argument of the `\useexternalfigure` command that associates an external figure with a symbolic name.
- At the exact moment when we insert a specific image. In this case the adjustment is made in the `\externalfigure` command itself.

The changes in the image that can be achieved by this route are the following:

Changing the size of the image. We can do this:

- *By assigning a precise width or height*, something done with the `width` and `height` options respectively; if only one of the two values is adjusted, the other is automatically adapted to maintain the proportion.

  We can assign a precise height or width, or indicate it as a percentage of page height or line width. For example:

```
width=.4\textwidth
```

will ensure that the image has a width equal to 40% of the line width.

- *Scaling the image*: The `xscale` option will scale the image horizontally; `yscale` will do so vertically, and `scale` will do it horizontally and vertically. These three options expect a number representative of the scaling factor multiplied by 1000. That is to say: `scale=1000` will leave the image in its original size, while `scale=500` will reduce it by half, and `scale=2000` will double its size.

  A conditional scaling, which is applied only if the image exceeds certain dimensions, is obtained with the `maxwidth` and `maxheight` options that take a dimension. For example `maxwidth=.2\textwidth` will scale the image only if it turns out to be more than 20% of the line width.

Rotating the image. To rotate the image we use the `orientation` option which takes a number representative of the number of degrees of rotation that will be applied. The rotation is done in a counter-clockwise direction.

> The wiki implies that the rotations that can be achieved with this option must be multiples of 90 (90, 180 or 270) but to achieve a different rotation we would have to use the `\rotate` command. However, I have not had any problem applying a 45 degree rotation to an image with only `orientation=45`, without the need to use the `\rotate` command.

Framing the image. We can also surround the image with a frame using the `frame=on` option, and configure its colour (`framecolor`), the distance between the frame and the image (`frameoffset`), the thickness of the line that draws the frame (`rulethickness`) or the shape of its corners (`framecorner`) which can be rounded (`round`) or rectangular.

Other configurable aspects of images. In addition to the aspects already seen, which imply a transformation in the image to be inserted, using `\setupexternalfigures` we can configure other aspects, such as where to look for the image (`directory` option), whether the image should be searched for only in the indicated directory (`location=global`) or whether it should also include the working directory and its parent directories (`location=local`), and whether the image will or will not be interactive (`interaction`), etc.

B. Specific commands for transforming an image

There are three commands in ConTEXt that produce some transformation in an image and can be used in combination with `\externalfigure`. These are:

- *Mirror image*: achieved with the `\mirror` command.
- *Clipping*: this is achieved with the `\clip` command when the width (`width`), height (`height`), horizontal offset (`hoffset`) and vertical offset (`voffset`) dimensions are given. For example:

```
\clip
  [width=2cm, height=1cm, hoffset=3mm, voffset=5mm]
  {\externalfigure[logo.pdf]}
```

- *Rotation.* A third command able to apply transformations to an image is the `\rotate` command. It can be used in conjunction with `\externalfigure` but normally this would not be necessary given that the latter has, as we have seen, the `orientation` option that produces the same result.

The typical use of these commands is with images, but they actually act on *boxes*. That's why we can apply them to any text simply by enclosing it in a box (which the command does automatically), that will produce curious effects like the following:

```
\mirror{Test text}\\
\rotate[rotation=20] {Test text}
```

Test text

Test text

13.3 Tables

13.3.1 General ideas about tables and their placement in the document

Tables are structured objects that contain text, formulas or even images arranged in a series of *cells* that allow us to graphically see the relationship between the contents of each cell. To do this, the information is organised into rows and columns: all data (or entries) in the same row have a certain relationship to each other, as well as all data (or entries) in the same column. A cell is the intersection of a row with a column, as shown in figure 13.3.

Tables are ideal for displaying text or data that are related to each other, because as each one is locked in its own cell, even if its content, or the content of the remaining cells changes, the relative position of one with respect to the others will not change.

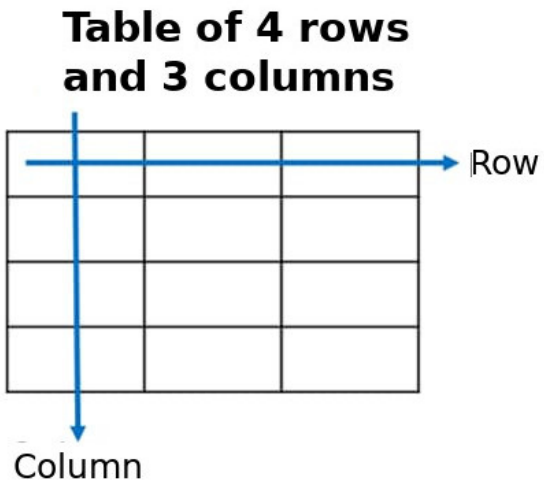


Figure 13.3 Image of a simple table

Of all the tasks involved in typesetting a text, the creation of tables is the only one that, in my opinion, is easier to do in a graphic program (word processor type) than in ConTEXt. Because it's easier *to draw* the table (which is what you do in a word processing program) than *describe it* which is what we do when we work with ConTEXt. Every row change and column change requires the presence of a command, and that means that it takes much longer to implement the table, instead of simply saying how many rows and columns we want.

ConTEXt has three different mechanisms for producing tables; the `tabulate` environment which is recommended for simple tables and which is the most directly inspired by TEX tables; the so-called *natural tables*, inspired by HTML tables, suitable for tables with special design needs where, for example, not all rows have the same number of columns; and the so-called *extreme tables*, clearly based on XML and recommended for medium or long tables which take up more than one page. Of the three, I will explain only the first. The natural tables are reasonably well explained in "ConTEXt Mark IV an excursion", and for *extreme tables* there is a document about them in the "ConTEXt Standalone" documentation.

Something similar to what happens with images occurs in tables: we can simply write the necessary commands at some point in the document to generate a table and it will be inserted at that exact point, or we can use the `\placetable` command to insert a table. This has some advantages:

- ConTEXt numbers the table and adds it to the list of tables allowing internal references to the table (through its numbering), including it in an eventual index of tables.
- We will gain flexibility in table placement within the document, thus facilitating the task of pagination.

The format of \placetable is similar to what we saw for \placefigure (see section 13.2.2):

```
\placetable[Options][Label] {Title} {table}
```

I refer to sections 13.4.1 and 13.4.2 regarding options relating to table placement and configuring the title. Among the options there is one, however, that seems to be designed exclusively for tables. This is the "split" option which, when set, authorises ConTEXt to extend the table over two or more pages, in which case the table can no longer be a floating object.

In general terms we can set the configuration for tables with the \setuptables command. Also, as with images, it is possible to generate an index of tables with \placelistoftables or \completelistoftables. In this regard see section 8.2.2.

13.3.2 Simple tables with the tabulate environment

The simplest tables are those achieved with the *tabulate* environment whose format is:

```
\starttabulate[Table column layout]
  ... % Table contents
  ...
  ...
\stoptabulate
```

Where the argument taken in square brackets describes (in code) the number of columns the table will have, and (sometimes indirectly) indicates their width. I say that the argument describes the design *in code*, because at first glance it seems very cryptic: it consists of a sequence of characters, each with a special meaning. I will explain it little by little and in steps, because I think that this way it is easier to understand.

> This is the typical case in which the huge number of aspects that we can configure means we need a lot of text to describe it. This seems to be devilishly difficult. In fact, for most of the tables that are built in practice, points 1 and 2 are enough. The rest are extra possibilities that it is useful to know exist, but are not essential to know to typeset a table.

1. **Columns delimiter**: the "|" character is used to delimit table columns. So, for example, "[|lT|rB|]" will describe a table with two columns, one of which would have the characteristics associated with the indicators "l" and "T" (which we will see immediately following) and the second column will have

the characteristics associated with "`r`" and "`B`". A simple three-column table aligned to the left, for example, would be described as "`[|l|l|l|]`".

2. **Determining the basic nature of the cells in a column:** The first thing to determine when we build our table is if we want the content of each cell to be written on a single line, or if, on the contrary, if the text of any column is too long we want our table to distribute it over two or more lines. In the `tabulate` environment that question is not decided cell by cell but is considered a characteristic of the columns.

   a. *One line cells:* If the contents of the cells in a column, regardless of their length, are to be written on a single line, we must specify the alignment of the text in the column, which can be left ("`l`", from *left*), right ("`r`", from *right*) or centred ("`c`", from *center*).

      In principle, these columns will be as wide as necessary to fit the widest cell. But we can limit the width of the column with the "`w(Width)`" specifier. For example, "`[|rw(2cm)|c|c|]`" will describe a table with two columns, the first aligned to the right and with an exact width of 2 centimetres, and the other two centred and with no width limitation.

      It should be noted that the width limitation in single-line columns may cause the text in one column to overlap text in the next column. So my advice is that when we need fixed-width columns, always use multi-line cell columns.

   b. *Cells that can take up more than one line if needed*: the "`p`" specifier generates columns in which the text in every cell will occupy as many lines as needed. If we simply specify "`p`", the width of the column will be the full width available. But it is also possible to indicate "`p(Width)`", in which case the width will be that expressly specified. Thus the following examples:

```
\starttabulate[|l|r|p|]
\starttabulate[|l|p(4cm)|]
\starttabulate[|r|p(.6\textwidth)|]
\starttabulate[|p|p|p|]
```

      The first example will create a table with three columns, the first and second of a single line, aligned, to the left and right respectively, and the third, which will occupy the remaining width and the height required to house all its contents. In the second example, the second column will measure exactly four centimetres wide, whatever its content; but if it does not fit in that space, it will take up more than one line. The third example calculates the width of the second column in proportion to the maximum width of the line, and in the last example, there will be three columns that will be the width available in equal parts.

      Note that, in reality, if a cell is a quadrilateral, what the "`p`" specifier does is authorise a variable height for the cells in a column, depending on the length of the text.

3. **Adding indications to the description of the column, about the font style and variant to be used**: once the basic nature of the column (width and height, automatic or fixed, of the cells) has been decided, we can still add, in the description of the contents of the column, a character representative of the *format* in which it must be written. These characters can be "`B`" for bold, "`I`" for italic, "`S`" for slanted, "`R`" for Roman style lettering or "`T`" for *typewriter* style lettering.

4. **Other additional aspects that can be specified in the description of table columns**

   :

   - *Columns with maths formulas*: the "`m`" and "`M`" specifiers enable maths mode in a column without the need to specify it in each of its cells. The cells in this column will not be able to hold normal text.

     > Although TEX, ConTEXt's predecessor, came into existence for typesetting any kind of maths, until now I have hardly said anything about writing maths. In the maths mode (which I will not be explaining) ConTEXt alters our normal rules and even uses different fonts. The maths mode has two varieties: one we could call *linear* in that the formula is housed within a line containing normal text ("`m`" indicator), and the *complete maths mode* that displays formulas in an environment where there is no normal text. The main difference between the two modes, in a table, is basically the size which the formula will be written in and the horizontal and vertical space surrounding it.

   - *Add extra horizontal white space around the contents of the cells in a column*: with the "`in`", "`jn`" and "`kn`" indicators we can add extra white space to the left of the column contents ("`in`"), to the right ("`jn`") or to both sides ("`kn`"). In all three cases "`n`" represents the number by which to multiply the white space that would normally be left without one of these specifiers (by default the average is an *em*). So, for example, "`|j2r|`" will indicate that we are faced with a column that will be aligned to the right, and in which we want a blank space of 1 *em*'s width.

   - *Adding text before or after the contents of each cell in a column*. The `b{Text}` and `a{Text}` specifiers cause the text between curly brackets to be written before ("`b`", from *before*) or after ("`a`", from *after*) the cell's contents.

   - *Applying a format command to the entire column*. The "`B`", "`I`", "`S`", "`R`" "`T`" indicators we mentioned previously do not cover all the format possibilities: e.g. there is no indicator for small caps, or for *sans serif*, or that

affects the font size. With the "`f\Command`" indicator we can specify a format command that is automatically applied to all cells in a column. For example, "`|lf\sc|`" will typeset the column's contents in small caps.

- *Applying any command to all the cells in the column.* Finally, the "`h\Command`" indicator will apply the specified command to all cells in the column.

In table 13.1 some examples of table format specification strings are shown.

Format specifier	**Meaning**
`|l|`	Generates a column whose width is automatically left-aligned.
`|rB|`	Generates a column whose width is automatically right-aligned, and in bold.
`|cIm|`	Generates a column enabled for maths content. Centred and in italics.
`|j4cb{---}|`	This column will have contents centred, will begin with an em dash (—) and will add 2 *ems* white space to the right.
`|l|p(.7\textwidth)|`	generates two columns: the first is left-aligned and width automatic. The second takes up 70% of the total width of the line.

Table 13.1 Some examples of how to specify the format of the columns in `tabulate`

Once the table has been designed, its contents need to be input. To explain how to do this I will start by describing how a table should be filled in where we have lines separating rows and columns:

- We always start by drawing a horizontal line. In a table this is done with the `\HL` command (from *Horizontal Line*).
- Then we write the first line: at the beginning of each cell we must indicate that a new cell begins and that a vertical line must be drawn. This is done with the `\VL` command (from *Vertical Line*). So we start with this command, and we write the content of each cell. Every time we change cells we repeat the `\VL` command.
- At the end of a row, we expressly indicate that a new row is going to be started with the `\NR` command (from *Next Row*). After it we repeat the `\HL` to draw a new horizontal line.
- And so, one by one, we write all the rows of the table. When we finish we add, as an extra, a `\NR` command and another `\HL` to close the grid with the bottom horizontal line.

If we do not want to draw the table grid, we remove the \HL commands and replace the \VL commands with \NC (from *New Column*).

It's not especially difficult when we get the hang of it, although when we look at the source code for a table it's hard to get an idea what it will look like. In table 13.2 we see the commands that can (and must) be used within a table. There are some that I have not explained, but I think the description I have given is enough.

Command	Meaning
\HL	Inserts a horizontal line
\NC	Begins a new column
\NR	Begins a new row
\VL	Inserts a vertical line delimiting a column (used in place of \NC)
\NN	Begins a column in maths mode (used in place of \NC)
\TB	Adds some extra vertical space between two rows
\NB	Indicates that the next row starts an indivisible block within which there cannot be a page break

Table 13.2 Commands to be used within a table

And now, as an example I will transcribe the code with which table 13.2 was written.

```
\placetable
  [here]
  [tbl:tablecommands]
  {Commands to be used within a table}
{\starttabulate[|l|p(.6\textwidth)|]
\HL
\NC {\bf Command}
\NC {\bf Meaning}
\NR
\HL
\NC \tex{HL}
\NC Inserts a horizontal line
\NR
\NC \tex{NC}
\NC Begins a new column
\NR
\NC \tex{NR}
\NC Begins a new row
\NR
\NC \tex{VL}
\NC Inserts a vertical line delimiting a column (used in place of \tex{NC})
\NR
\NC \tex{NN}
```

```
\NC Begins a column in maths mode (used in place of \tex{NC})
\NR
\NC \tex{TB}
\NC Adds some extra vertical space between two rows
\NR
\NC \tex{NB}
\NC Indicates that the next row starts an indivisible block within which there
cannot be a page break
\NR
\HL
\stoptabulate}
```

The reader will notice that in general I have used one (or two) lines of text for each cell. In a real source file I would have only used a line of text for each cell; in the example I have split the lines that are too long. Using a single line per cell makes it easier for me to write the table because what I do is to write the contents of each cell, without row or columns separation commands. When everything is written, I select the text from the table and ask my text editor to insert "`\NC` " at the beginning of each line. After that, every two lines (because the table has two columns) I insert a line that adds the `\NR` command, because every two columns starts a new row. Finally, by hand, I insert the `\HL` commands at the points where I want a horizontal line to appear. It takes me almost longer to describe it than to do it!

But also see how, within a table, we can use ConTEXt's ordinary commands. In particular in this table we continually use `\tex` which is explained in section 10.2.3.

13.4 Aspects common to images, tables and other floating objects

We already know that images and tables do not have to be floating objects, but they are good candidates to be so, although they have to be inserted in the document by means of the `\placefigure` or `\placetable` commands. In addition to these two commands, and with the same structure, in ConTEXt we have the `\placechemical` command (to insert formulas chemicals), the `\placegraphic` command (to insert graphics) and the `\placeintermezzo` command for inserting a structure that ConTEXt calls *Intermezzo* and which I suspect refers to framed text fragments. All these commands are in turn concrete applications of a more general command that is `\placefloat` whose syntax is the following:

```
\placefloat[Name] [Options] [Label] {Title} {Contents}
```

Note that `\placefloat` is identical to `\placefigure` and `\placetable` except for the first argument that in `\placefloat` takes the name of the floating object. This is because *each type of floating object can be inserted into the document with*

two different commands: `\placefloat[TypeName]` or `\placeTypeName`. In other words: `\placefloat[figure]` and `\placefigure` are exactly the same command, just as `\placefloat[table]` is the same command as `\placetable`.

I will therefore speak from now on of `\placefloat`, but please note that everything I say will also apply to `\placefigure` or `\placetable` which are specific applications of said command.

The `\placefloat` arguments are:

- *Name*. refers to the floating object in question. It can be some predetermined floating object (`figure, table, chemical, intermezzo`) or a floating object created by ourselves using `\definefloat` (see section 13.5).
- *Options*. A series of symbolic words that tell ConTEXt how it should insert the object. The great majority of these refer to *where* to insert it. We will see this in the next section.
- *Label*. A label for future internal references to this object.
- *Title*. The title text to be added to the object. Regarding its configuration, see section 13.4.2.
- *Contents*. This depends, of course, on the type of object. For images it is usually a `\externalimage` command; for tables, the commands that will create the table; for *intermezzi*, a framed text fragment; etc.

The first three arguments, which are introduced in square brackets, are optional. The last two (which are introduced between curly brackets) are mandatory, although they can be empty. So, for example: `\placefloat{}{}` will insert:

Figure 13.4

in the document.

Note: We see that ConTEXt considered that the object to be inserted was an image, since it was numbered as an image and included in the list of images. This makes me assume that images are the default floating objects.

13.4.1 Floating object insertion options

The *Options* argument in `\placefigure`, `\placetable` and `\placefloat` controls different aspects regarding the insertion of these types of objects. Mainly the place on the page where the object will be inserted. Here several values are supported, each of a different nature:

- Some of the insertion places are established in relation to page elements (`top, bottom inleft, inright, inmargin, margin, leftmargin, rightmargin, leftedge, rightedge, innermargin, inneredge, outeredge, inner, outer`). It must, of course, be an object that can fit in the area where it is intended to be placed and space must have been reserved for that element in the page layout. Regarding this, see section 5.2 and 5.3.

- Other possible insertion places are more related to the text surrounding the object, and are an indication of where the object should be placed so that the text flows around it. Fundamentally the `left` and `right` values.

- The `here` option is interpreted as a recommendation to keep the object at the point in the source file where it is located. This *recommendation* will not be respected if the pagination requirements do not allow it. This indication is reinforced if we add the `force` option which means exactly that: force the insertion of the object at that point. Note that by forcing the insertion at a particular point, the object will no longer be floating.

- Other possible options relate to the page on which the object is to be inserted: "`page`" inserts it on a new page; "`opposite`" inserts it on the page opposite the current one; "`leftpage`" on an even page; "`rightpage`" on an odd page.

There are some options that are not related to the location of the object. Among them:

- `none`: This option suppresses the title.

- `split`: This option allows the object to extend over more than one page. It must, of course, be an object that is divisible by nature, such as a table. When this option is used and the object is split, it can no longer be said to be floating.

13.4.2 Configuring floating object titles

Unless we use the "`none`" option in `\placefloat`, by default, floating objects are associated with a title that consists of three elements:

- The name of the type of object in question This name is exactly that of the object type; so if, for example, we define a new floating object called "sequence"

and we insert a "sequence" as a floating object, the title will be "Sequence 1". Simply capitalise the name of the object.

> Despite what has just been said, if the main language of the document is not English, the English name for predefined objects, like for example the "`figure`" or "`table`" objects, will be translated; So, for example, the "`figure`" object in documents in Spanish are called "`Figura`", while the "`table`" object is called "`Tabla`". These Spanish names for predefined objects can be changed with `\setuplabeltext` as explained in section 10.5.3.

- Its number. By default the objects are numbered by chapters, and so the first table in Chapter 3 will be table '3.1'.
- Its contents. Introduced as an argument of `\placefloat`.

With `\setupcaptions` or `\setupcaption[Object]` we can change the numbering system and the appearance of the title itself. The first command will affect all the titles of all objects, and the second will affect only the title of a particular type of object:

- As for the numbering system, this is controlled by the `number`, `way`, `prefixsegments` and `numberconversion` options:
  - `number` can adopt the `yes`, `no` or `none` values and controls whether there will be a number or not.
  - `way` indicates whether the numbering will be sequential throughout the document (`way=bytext`), or whether it will recommence at the beginning of each chapter (`way=bychapter`) or section (`way=bysection`). In the case of a restart, it is appropriate to coordinate the value of this option with the `prefixsegments` option.
  - `prefixsegments` indicates if the number will have a *prefix*, and what this will be. Thus `prefixsegments=chapter` causes the number of objects to always start with the chapter number, while `prefixsegments=section` will precede the object number with the section number.
  - `numberconversion` controls the kind of numeration. The values for this option can be: Arabic numbers ("`numbers`"), lower case ("`a`", "`characters`"), upper case ("`A`", "`Characters`"), small caps "`KA`"), upper case Roman numerals ("`I`", "`R`", "`Romannumerals`"), lower case ("`i`", "`r`", "`romannumerals`" or small caps ("`KR`")).
- The appearance of the title itself is controlled by numerous options. I will list them, but for a detailed explanation of the meaning of each one of them, I refer to section 7.4.4 where the control of the appearance of the sectioning commands is explained, as the options are largely the same. The options in question are:

  - To control the format of all the elements of the title, `style, color, command`.
  - To control the format only of the name for the kind of object: `headstyle, headcolor, headcommand, headseparator`.
  - To control only the numbering format: `numbercommand`.
  - To control only the format of the title itself: `textcommand`.

- We can also control other aspects such as the distance between the different elements that make up the title, the width of the title, its placement in relation to the object, etc. I refer here to the information in ConTEXt wiki regarding the options that can be configured with this command.

13.4.3 Combined insertion of two or more objects

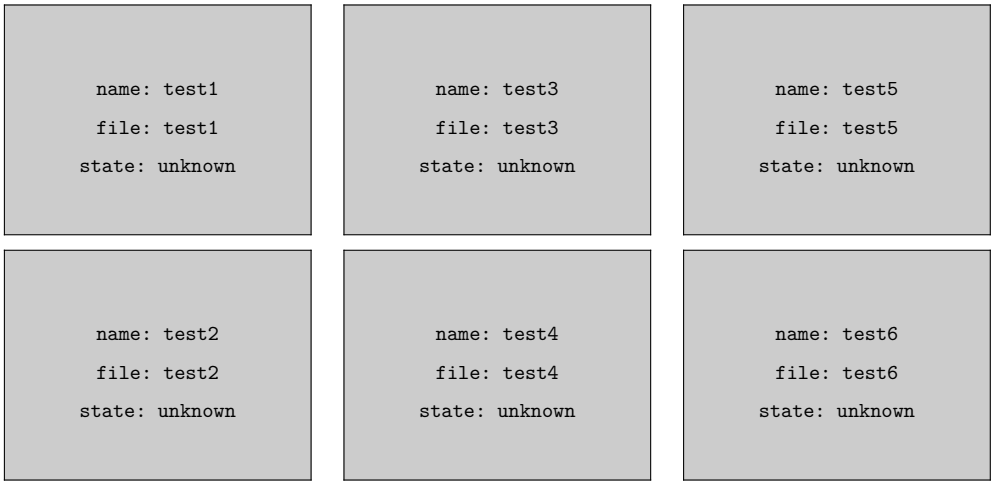
To insert two or more different objects in the document, such that ConTEXt keeps them together and deals with them as a single object, we have the `\startcombination` environment whose syntax is:

```
\startcombination[Ordering] ... \stopcombination
```

where *Ordering* indicates how the objects should be ordered: if they all need to be ordered horizontally, *Ordering* only indicates the number of objects to be combined. But if we want to combine the objects in two or more rows, we will have to indicate the object number per row, followed by the number of rows, and separating both numbers by the * character. For example:

```
\startcombination[3*2]
  {\externalfigure[test1]}
  {\externalfigure[test2]}
  {\externalfigure[test3]}
  {\externalfigure[test4]}
  {\externalfigure[test5]}
  {\externalfigure[test6]}
\stopcombination
```

which will produce the following alignment of images.



In the previous example, the images I have combined actually do not exist, which is why, instead of the images, ConTEXt has generated text boxes with information about them.

See, on the other hand, how each element to be combined within `\startcombination`, is enclosed within curly brackets.

In fact, `\startcombination` not only allows us to connect and align images, but any kind of *box* such as texts inside a `\startframedtext` environment, tables, etc. To configure the combination we can use the `\setupcombination` command and we can also create pre-configured combinations using `\definecombination`.

13.4.4 General configuration of floating objects

We have already seen that with `\placefloat` we can control the location of the floating object being inserted and some other details. It is also possible to configure:

- The global characteristics of a particular type of floating object. This is done with `\setupfloat[Name of type of floating object]`.
- The global characteristics of all floating objects in our document. This is done with `\setupfloats`.

Bear in mind that in the same way that `\placefloat[figure]` is equivalent to `\placefigure`, `\setupfloat[figure]` is equivalent to `\setupfigures`, and `\setupfloat[table]` is equivalent to `\setuptables`.

Regarding the configurable options for these, I refer to the ConTEXt official list of commands (section 3.6).

13.5 Defining additional floating objects

The `\definefloat` command allows us to define our own floating objects. Its syntax is:

```
\definefloat[Singular name] [Plural name] [Configuration]
```

Where the *Configuration* argument is an optional argument that allows us to already indicate the configuration of this new object at the time of its creation. We can also do it later with `\setupfloat[Name in the singular]`.

Since we are ending our introduction with this section, I am going to take advantage of it to go a little deeper into the apparent *jungle* of ConTEXt commands which, once understood, is not so much of a *jungle* but is, in fact, quite rational.

Let's start by asking ourselves what a floating object really is forConTEXt, the answer being that it is an object with the following characteristics:

- That it has a certain margin of freedom with respect to its location on the page.
- That is has a *list* associated with it, that allows it to number these kinds of objects and, eventually, to generate an index of them.
- That it has a title
- That, when the object can really float, it must be treated as an inseparable unit, meaning (in TEX terminology) *enclosed in a box*.

In other words, the floating object is actually made up of three elements: the object itself, the list associated with it, and the title. To control the object itself we only need one command to set up its location and another to insert the object into the document; to set up the list aspects, general list control commands are sufficient, and to set up the title aspects, the general title control commands.

And this is where the genius of ConTEXt comes in: a simple command to control floating objects (`\setupfloats`), and a simple command to insert floating objects: `\placefloat`, could have been designed: but what ConTEXt does is to:

1. Design a command to link a name to a specific floating object configuration. This is the `\definefloat` command, which does not actually link one name, but two names, one in the singular and one in the plural.
2. Create, together with the global floating objects configuration command, a command that allows us to configure only a specific type of object: `\setup-float[Object]`.

3. Add to the floating object location command, (\placefloat), an argument that allows us to differentiate between one or other type: (\placefloat[Object]).

4. Create commands, including the object name, for all actions of a floating object. Some of these commands (which are actually clones of other more general commands) will use the object's name in the singular and others will use it in the plural.

Therefore, when we create a new floating object and tell ConTEXt what its name is in the singular and the plural, ConTEXt:

- Reserves a space in memory to store the specific configuration of that object type.
- Creates a new list with the singular name of that object type, since floating objects are associated with a list.
- Creates a new kind of "title" linked to this new object type, in order to maintain a customised configuration of these titles.
- And finally, it creates a group of new commands specific to that new object type, whose name is actually a synonym for the more general command.

In table 13.3 we can see the commands that are automatically created when we define a new floating object, as well as the more general commands they are synonyms of:

Command	Synonym of	Example
\completelistof<PluralName>	\completelist[PluralName]	\completelistoffigures
\place<SingularName>	\placefloat[SingularName]	\placefigure
\placelistof<PluralName>	\placelist[PluralName]	\placelistoffigures
\setup<SingularName>	\setupfloat[SingularName]	\setupfigure

Table 13.3 Commands that are automatically created when a new floating object is created

Actually, some additional commands are created which are synonymous with the previous ones and since I have not included them in the explanation of the chapter, I have omitted them from table 13.3: \start<NameSingular>, \start<NameSingular>text and \startplace<NameSingular>.

I have used the command used for images as an example of the commands created when defining a new floating object; and I did so because images, like tables and the rest of the floats predefined by ConTEXt, are actual cases of \definefloat:

```
\definefloat[chemical][chemicals]
\definefloat[figure][figures]
```

```
\definefloat[table][tables]
\definefloat[intermezzo][intermezzi]
\definefloat[graphic][graphics]
```

Finally, we see that in reality ConTEXt in no way controls any kind of material included in each particular floating object; it presumes that this is the author's job. This is why we can also insert text with the `\placefigure` or `\placetable` commands. However, the text that is input with `\placefigure` is included in the list of images, and if input with `\placetable`, in the list of tables.

Appendices

Appendix A
Installing, configuring and updating ConTEXt

TEX's main distributions (TeX Live, teTeX, MikTeX, MacTeX, etc.) include a version of ConTEXt. However, this is not the most updated version. In this appendix I will explain two procedures to install two different versions of ConTEXt; the first includes both ConTEXt Mark II and Mark IV and the second includes only ConTEXt Mark IV.

The installation procedure follows the same steps on any operating system; but the details change from one system to another. However, we can simplify things in such a way that in the following lines I will distinguish between two big groups of systems:

- **Unix-type systems:** This includes Unix itself, as well as GNU Linux, Mac OS, FreeBSD, OpenBSD or Solaris. The procedure is basically the same in all these systems; there are some very small differences that I will highlight in the appropriate place.
- **Windows systems**, that includes the different versions of that operating system: Windows 10 (the latest version, I think), Windows 8, Windows 7, Windows Vista, Windows XP, Windows NT, etc.

  Important note on the installation process on Microsoft Windows systems:

  ConTEXt, like all TEX systems, is designed to work from a terminal; the programs and procedures for installation, too. In Windows this is also perfectly possible and should not create any major difficulty. The problem is that, on the one hand Windows users are not always used to doing this, and on the other, since Windows came into being in the *illusion* (false) that everything in a computer system could be done graphically, in general the versions of that operating system do not *advertise* too much about how to use the terminal. And then, it is common for each version of this system to change the name of the program that runs the terminal and how to open it. As far as I know, the Windows terminal emulation program has been given many names: "DOS window", "Command Prompt", "cmd", etc. The location of this program in the Windows application menu also changes depending on the version of Windows in question.

  I stopped using Windows-based systems in 2004, so there is little I can do here to help the reader. He or she will have to figure out, on their own, how to open a terminal in their particular version of the operating system; which shouldn't be too difficult.

1 Installing and configuring "ConTEXt Standalone"

The ConTEXt distribution known as "Standalone", also known as "ConTEXt Suite", is a complete and updated distribution of ConTEXt, which downloads the necessary files from the Internet, does not take up too much disk space, is easy to update, and above all — hence the name *Standalone* — is contained in a single directory which can be located anywhere we want on the hard disk. It would even be possible for a single computer to have several versions of ConTEXt each in its own directory. This distribution includes the fonts, binary files and documentation needed to run ConTEXt Mark II (which implies the TEX PdfLatex and XeTeX engines), and ConTEXt Mark IV (which implies the LuaTeX engine).

> For information about TEX *engines*, see section 1.4.1; and on TEX engines in relation to ConTEXt, as well as the versions known as Mark II and Mark IV, section 1.5.1.

The following explains how to install, run, update and restore "ConTEXt Standalone" on our system. The data and procedures provided here are a summary of the much more extensive information included in the ConTEXt wiki, to which I have added some additional detail drawn from a wikibook on ConTEXt hosted on wikibooks. If there is any problem with the installation, or if you want to extend any detail, you should directly consult any of these (though the latter is in French)

1.1 Installation

Installing "ConTEXt Standalone" means having an Internet connection, and implies the following steps:

1. Creating the directory in which ConTEXt will be installed.
2. Downloading the installation *script* into this directory.
3. Running this *script* with the desired options.
4. Making some final adjustments.

Step 1: creating the installation directory

This, in fact, has nothing to do with ConTEXt and we have to assume that every user will know how to do it. In Windows systems the normal way is to do it from the file manager. On Unix-type systems, it can be done from a file manager or from a terminal. It is important, however, to keep in mind that it is not recommended that the installation directory contains any blank space in your path. I personally also tend to shy away from using non-English directory names with things like accented vowels in them.

From now on I will assume that the installation directory is, in Unix-like systems, "`~/context/`" and in Windows, "`C:\Programs\context`".

Step 2: Download the installation *script* into the installation directory

The installation *script* will differ according to the operating system you are installing on:

- On Unix-like systems it can be downloaded, with a web browser, or, from a terminal with "`wget`" or "`rsync`":

```
wget http://minimals.contextgarden.net/setup/first-setup.sh
rsync rsync://minimals.contextgarden.net/setup/first-setup.sh
```

- On Windows-type systems, as far as I know, there are no standard tools for downloading from the console. It has to be done with a web browser. The download address can be any of the following:

  http://minimals.contextgarden.net/setup/context-setup-mswin.zip
  http://minimals.contextgarden.net/setup/context-setup-win64.zip

  Once downloaded, in Windows you have to unzip the file,

Step 3: Run the installation *script*

The installation *script* must be run from the terminal. In Unix-type systems the name of the *script* is "`first-setup.sh`" and can be run with `bash` or `sh`. In Windows-type systems the *script* is called "`first-setup.bat`" and is run by simply typing its name in the system console or MS-DOS window from the installation directory.

The installation *script* allows for the following options:

- `--context`: this option determines which version of ConTeXt will be installed, whether the most recent development version ("–context=latest") or the latest stable version ("–context=beta"). The default value is "beta".
- `--engine`: allows us to indicate whether we want to install Mark IV ("–engine=luatex", the default value) or Mark II.
- `--modules`: also install the ConTeXt extension modules that do not belong to the distribution as such, but that offer interesting additional utilities. To do this we need to indicate "–modules=all".

  With regard to the installation options, I believe that the information in the wiki is now obsolete. There it says that to install only Mark IV you need to explicitly indicate the "–engine=luatex"

option and that the "–context=latest" option installs the latest stable version, not the development version. However, from halfway through 2020 the content of first-setup.sh changed, and taking a look inside it I found that to install the very latest version you need to expressly indicate "–context=latest", and that "–engine=luatex" is enabled by default.

The French Wikibook I mentioned at the beginning adds two other possible options to the options I just mentioned (documented on the ConTEXt wiki): "–fonts=all" and "goodies=all". ConTeXtgarden doesn't mention them, but including them in the installation command as well doesn't hurt. Therefore I would advise you to run the installation script with the following options (depending on whether we are on a Unix- or Windows-type system):

- Unix: `bash first-setup.sh --context=latest --modules=all --fonts=all --goodies=all`
- Windows: `first-setup.bat --context=latest --modules=all --fonts=all --goodies=all`

This, depending on the speed of our Internet connection, may take some time, but not too much.

Configuring a proxy

The installation script uses rsync to obtain the necessary files. So, if you are behind a proxy server, you need to specify its details to rsync. The easiest way to set this is to set the variable RSYNC_PROXY in the terminal or in your startup *script* (.bashrc or the corresponding file for each shell). Replace the username, password, proxyhost and proxyport with the correct information. This is done, on Unix-type systems, with the "`export`" command, and in Windows-type systems with the "`set`" command. For example:

```
export RSYNC_PROXY=username:password@proxyhost:proxyport
```

Sometimes, when we are behind a firewall, port 873 may be closed for outgoing TCP connections. If port 22 is open for ssh connections, one trick that can be used is to connect to a computer somewhere outside the firewall and tunnel into port 873 (using the nc program).

```
export RSYNC_CONNECT_PROG='ssh tunelhost nc %H 873'
```

where the 'tunnelhost' is the machine outside the firewall we have access to. Of course, this machine must have nc and port 873 open for the outgoing TCP connection

After running "`first-setup`" in the installation directory two new directories will appear called, "`bin`" and "`tex`" respectively.

Step 4: Final adjustments (Only on GNU Linux)

In GNU Linux systems there are many directories where fonts can be installed. If we want ConTEXt to use these fonts we must tell it where to find them. To do this we must add the following line to the "`tex/setuptex`" file created after the installation:

```
export OSFONTDIR="~/.fonts:/usr/share/fonts:/usr/share/texmf/fonts/opentype/"
```

with which the environment variable OSFONTDIR is loaded with the three directories in which the fonts installed in the system are normally located

> The /usr/share/texmf/fonts/ will only be there if there is some other installation of TEX or other systems based on it in our operating system; in this case it should be included in the OSFONTDIR path so we can use the opentype fonts that such an installation may have included. If you have any commercial fonts that you want ConTEXt to use, you have to make sure that the path to these is one of those included in OSFONTDIR, or otherwise, add the path to this variable. I have seen, for example, that some fonts are installed in /usr/local/fonts instead of /usr/share/fonts.

Finally, it may be a good idea to have ConTEXt generate a database with the necessary files for execution. This will be done by running the following three commands from a terminal:

```
. ~/context/tex/setuptex
context --generate
context --make
```

The first instruction is a point (dot). That's an abbreviation for bash's internal `source` command. We can also, of course, run *source* if it's more convenient for us.

1.2 Running "ConTEXt Standalone"

"ConTEXt Standalone" has been designed to be able to coexist with other installations of TEX systems, which is an advantage because it allows us to have several different versions installed on the same operating system; but in order to exploit this advantage it is essential that the environment variables needed to run ConTEXt are not set permanently, because every time we start a terminal to run "`context`" from it, we'll have to start by loading these environment variables into memory. They are contained in the "`tex/setuptex`" (Unix) or "`tex/setuptex.bat`" (Windows) file. This is done:

- In Unix-type systems, after opening the terminal in which we want to use "`context`", by running either of the following two commands:

  ```
  source ~/context/tex/setuptex
  . ~/context/tex/setuptex
  ```

  (assuming that the directory where the version of "`context`" we want to use is "`~/context`").

- In Windows-type systems, by running the `tex\setuptex.bat` command from the installation directory in the terminal from which we will use ConTEXt.

If there is no other installation of TEX or any of its derivatives in our system, we can avoid this by automating the execution of this order every time a terminal is opened:

- On Unix-like systems this is done by including it in the file containing the general terminal startup *script* (usually "`.bashrc`").

  The configuration file of a terminal depends on the *shell* program that the terminal uses by default. If this is `bash` (which is the most used in GNU Linux systems), the file read at the beginning is `.bashrc`. The `sh` and `ksh` *shells* use a file called `.profile`, `zsh` uses `.zshenv`, and `tcsh` or `csh` read the `.cshrc` file. Some specific implementation may change the names of these files and so, for example, `.bashrc` is sometimes called `.bash_profile`.

- In Windows-type systems we can create a shortcut on the desktop that runs cmd.exe and then edit it, putting as a command to run when we double click on it:

  ```
  C:\WINDOWS\System32\cmd.exe /k C:\Programs\context\tex\setuptex.bat
  ```

Another possibility, if we do not wish to run this script each time we want to use ConTeXt, nor want to permanently set the environment variables necessary for it to be run, is to do it from the text editor itself, instead of running ConTeXt from a terminal. How you do this depends on the particular text editor you are using. The ConTeXt wiki provides information on how to set up various common editors: LEd, Notepad++, Scite, TeXnicCenter, TeXworks, vim and some others.

1.3 Updating the version of "ConTEXt Standalone" or returning to an earlier version

Mark IV is still under development, so "ConTEXt Standalone" is often updated. To update our installation just repeat the process: we download a new version of "`first-setup.sh`" and run it.

If, for whatever reason, we want to go back to a previous version of "ConTEXt Standalone", just run "`first-setup`" with the "`--context=date`" option, where *date* is the date corresponding to the version we want to recover. Note that the date has to be introduced in the US months-days-years format.

The complete list of ConTEXt versions and associated dates can be found at this link.

Finally, keep in mind that after reinstalling the system, whether it is to upgrade or to return to a previous version, on GNU Linux systems you will have to run step 4 of the installation again, which I have called "Final Adjustments".

2 Installing LMTX

If we only plan to use ConTEXt Mark IV, and we want to compile our projects not directly with LuaTeX but with LuaMetaTex, a simplified LuaTeX that uses less

system resources and that can work on *less powerful* systems, instead of "ConTEXt Standalone", we need to install LMTX which is the latest version of ConTEXt. The name is an acronym of the name of the TEX engine being used: LuaMetaTeX. This version was launched in 2019, and since approximately May 2020 it is the recommended default ConTEXt distribution as suggested in ConTEXt wiki.

> The current development of LMTX is intense, and the beta version can change several times a week. Some of its developments, moreover, temporarily pose certain incompatibilities with Mark IV, and so, for example, while I am writing these lines, the latest version of LMTX (August 4, 2020) produces an error with the `\Caps` command. Therefore I would advise newcomers, for the moment, to work with "ConTEXt Standalone" instead.

2.1 The installation itself

The installation is as simple as:

- **Step 1:** Decide on the directory you want to install LMTX in, and, if necessary, create it. I will assume that the installation is done in a directory called "context" located in our user directory.
- **Step 2:** Download (to the installation directory) the zip file from the ConTEXt wiki that corresponds to your operating system and processor. It can be any of the following:
  - GNU/Linux
    - ⋆ X86 Processor
      - ▷ 32 bit version.
      - ▷ 64 bit version.
    - ⋆ ARM Processor
      - ▷ 32 bit version.
      - ▷ 64 bit version.
  - Microsoft Windows
    - ⋆ 32 bit version
    - ⋆ 64 bit version
  - Mac OS, versión de 64 bits
  - FreeBSD
    - ⋆ 32 bit version.
    - ⋆ 64 bit version.
  - OpenBSD6.6
    - ⋆ 32 bit version.
    - ⋆ 64 bit version.
  - OpenBSD6.7
    - ⋆ 32 bit version.
    - ⋆ 64 bit version.

If you don't know whether your system is 32-bit or 64-bit, chances are – unless your computer is very old – it's 64-bit. If you don't know whether your processor is X86 or ARM, it's most likely X86.

- **Step 3:** Unzip, the file downloaded in the previous step into the installation directory. A folder will be created called "`bin`" and two files, one called "`installation.pdf`", that contains more detailed information about the installation, and a second file which is the actual installation program called "`install.sh`" (in Unix-type systems) or "`install.bat`" (in Windows systems).

- **Step 4:** Run the installation program ("`install.sh`" or "`install.bat`"). It needs an Internet connection as the installation program searches the web for the files it needs.

  – On Unix-type systems the installation program, located in the installation directory, is run from a terminal, either with `bash`, or with `sh`. It is not necessary to have administrator privileges, unless the installation directory is outside the user's "`home`" directory.

  – In Windows-type systems, you must open a terminal, move to the installation directory, and from the terminal, run `install.bat`. It is not necessary here either that the installation program is run as a system administrator, but it is recommended that this be done so that symbolic links of the files can be used, thus saving disk space.

- **Step 5** inform the system of the path to LMTX:

  In Windows systems, the installation program generates a file, called "`setpath.bat`" which updates all the configuration files necessary to let Windows know that you have installed LMTX in the system and where you has done so. In GNU Linux systems, FreeBSD or Mac OS no *script* that automates the task is generated, so we must incorporate the address for the ConTEXt binaries in the system's PATH variable, which we would get by running in the terminal, from the installation:

  ```
  export PATH="InstallationDir/tex/texmf-Platform/bin:"$PATH
  ```

  where *`InstallationDir`* is the installation directory (for example, "/home/user/context") and `texmf-`*`Platform`* will vary according to the version of LMTX we have installed. For example, an installation on a 64 bit Linux system, `texmf-`*`Platform`* will be "texmf-linux-64". Therefore we should run the following command from the terminal:

  ```
  export PATH="/home/user/context/texmf-linux-64/bin:"$PATH
  ```

  This command will include LMTX in the system path, only as long as the terminal from which it has been run remains open. If we want this to be done

automatically every time a terminal is opened, we must include this command in the configuration file of the *shell* program used by default in the system. The name of this file changes according to which *shell* program it is: bash, sh, zsh, ksh, tcsh, csh... On most Linux systems, which use bash, the file is called "`.bashrc`" so we should run the following command from our home directory:

```
echo 'export PATH="/home/user/context/texmf-linux-64/bin:"$PATH >> .bashrc
```

Important note: By executing this step, we will disable the possibility of using other versions of ConTEXt on our system, such as the one incorporated in TeX Live or "ConTEXt Standalone". If we want to make both versions compatible, it is preferable to use the procedure described in section 3.

2.2 Installing extension modules in LMTX

ConTEXt LMTX does not incorporate a procedure for installing or upgrading the ConTEXt extension modules. However, in ConTEXt wiki there is a *script* that allows you to install and update all the modules along with the rest of the installation.

To do this we need to copy the aforementioned *script*, paste it into a text file located in the main LMTX installation directory (the one containing `install.sh` or `install.bat`) and run it from a terminal. I have personally verified that this works on a GNU Linux system. I'm not sure if it will work on a Windows system, since I don't have any version of that operating system to check it with.

2.3 Updating LMTX

Updating LMTX is as simple as running the installation program again: it will check the installed files against those on the web server and update as necessary.

If the website from which the files are obtained has changed, we obviously need to also change this address in the installation *script*; although perhaps it is easier to download a new version of the installation files in the same directory and extract from it the new "`install.sh`" or "`install.bat`"; or, even easier, unzip the file with the installation program and reinstall without first needing to removing the old files.

2.4 Creating a file that loads the variables into memory needed for LMTX (only GNU/Linux systems)

"ConTEXt Standalone" contains, as we already know, a ("`tex/setuptex`") file that loads into memory all the variables needed to run it, but LMTX does not include a similar file. We can, however, easily create it ourselves and store it, for example,

as "setuplmtx" in the "tex" directory. The commands that this file could have would be:

```
export PATH="~/context/LMTX/tex/texmf-linux-64/bin:"\$PATH
echo "Adding ~/context/LMTX/tex/texmf-linux-64/bin to PATH"
export TEXROOT="~/context/LMTX/tex"
echo "Setting ~/context/LMTX/tex as TEXROOT"
export OSFONTDIR="~/.fonts:/usr/share/fonts:/usr/share/texmf/fonts/opentype/"
echo "Loading font directories into memory"
alias lmtx="~/context/LMTX/tex/texmf-linux-64/bin/context"
echo "Creating an alias to run lmtx"
```

With this, besides loading into memory the paths and variables needed to run LMTX we would be enabling the "lmtx" command as a synonym of "context".

After creating this file, before being able to use LMTX, where we intend to use it we should run the following in the terminal:

```
source ~/context/LMTX/tex/setuplmtx
```

all this assuming that LMTX is installed in " /context/LMTX" and that we have called this file "setuplmtx" and stored it in " /context/LMTX/tex".

> The above is what I do, to work with LMTX in the same way I used to work with "ConTEXt Standalone". However, I do not exclude the possibility that in LMTX it is not necessary, for example, to load into memory the variable OSFONTDIR, since I am struck by the fact that ConTEXt wiki says nothing about this.

3 Using several versions of ConTEXt on the same system (only for Unix-type systems)

The operating system utility called alias allows us to associate different names with different versions of ConTEXt. So we can use, for example, the version of ConTEXt included in TeX Live and LMTX; or the *Standalone* version and LMTX.

For example, if we store the versions of LMTX downloaded in January and August 2020 in different directories, we could write the following two instructions in ".bashrc" (or equivalent file read by default when opening a terminal):

```
alias lmtx-01="/home/user/context/202001/tex/texmf-linux-64/bin/context"
alias lmtx-08="/home/user/context/202008/tex/texmf-linux-64/bin/context"
```

These instructions will associate the names lmtx-01 with the version of LMTX installed in the "context/202001" directory and lmtx-08 with the version installed in "context/202008".

Appendix B
Commands for generating maths and non-maths symbols

In the following tables you will find the commands that generate a variety of symbols, checked one by one by myself; most of them (but not all) are preferably for use in mathematics.

I am aware that how they are arranged could well be open to improvement. The problem is that since I am more of a literature background, I do not know what many of these symbols are used for in maths; and often I'm not even sure if it's really a case of symbols used in maths. That's why I made a group of the symbols that I am reasonably sure are not used for mathematics, and where the rest are concerned I grouped the different symbols according to certain recognisable shapes (triangles, squares, asterisks, rhombuses, arrows, points). I have arranged the rest of the symbols, *probably* they are ones used in maths, alphabetically (from the command that generates them).

The tables, moreover, are not exhaustive. I am sure that there are many more symbols that can be generated with ConTEXt, although I have not found any document or web page that collects them all. Those collected here are, for the most part, symbols that work in TEX or in LATEX, and that I have verified as also working in ConTEXt. Many of these symbols only work in maths mode in LATEX (enclosed in '$'). In ConTEXt, as is easy to see in the tables that follow, that is only necessary in very few cases.

Money and symbols for legal use

©	\copyright	®	\registered	¢	\textcent
℗	\textcircledP	¤	\textcurrency	$	\textdollar
₫	\textdong	€	\texteuro	f	\textflorin
£	\textsterling	¥	\textyen	™	\trademark

Triangles, círcles, squares and other shapes

△	\triangle	◯	\bigcirc	□	\square
◁	\triangleleft	∘	\circ	■	\blacksquare
▷	\triangleright	•	\bullet	⊡	\boxdot
▽	\triangledown	⊛	\circledast	⊟	\boxminus
▼	\blacktriangledown	◎	\circledcirc	⊞	\boxplus
◀	\blacktriangleleft	⊖	\circleddash	⊠	\boxtimes
▶	\blacktriangleright	⨁	\bigoplus	∗	\ast
▲	\blacktriangle	⨂	\bigotimes	✠	\maltese
≜	\triangleq	⊕	\oplus	⋆	\star
⋄	\diamond	⊖	\ominus	♣	\clubsuit
◊	\lozenge	⊗	\otimes	♡	\heartsuit
♦	\blacklozenge	⊘	\oslash	♠	\spadesuit
♢	\diamondsuit	⊙	\odot	∅	\varnothing

Arrows:

←	\leftarrow, \gets	→	\rightarrow, \to	↔	\leftrightarrow
↚	\nleftarrow	↛	\nrightarrow	⇔	\Leftrightarrow
⟵	\longleftarrow	⟶	\longrightarrow	⟷	\longleftrightarrow
⇐	\Lleftarrow	⇛	\Rrightarrow	⟺	\Longleftrightarrow
⇍	\nLeftarrow	⇏	\nRightarrow	⇋	\leftrightharpoons
↰	\Lsh	↱	\Rsh	↭	\leftrightsquigarrow
↤	\mapsfrom	↦	\mapsto	⇎	\nLeftrightarrow
⟻	\longmapsfrom	⟼	\longmapsto	↮	\nleftrightarrow
⇤	\Mapsfrom	⇨	\Mapsto	⇄	\rightleftarrows
⟽	\Longmapsfrom	⟾	\Longmapsto	⇌	\rightleftharpoons
↢	\leftarrowtail	↣	\rightarrowtail	↕	\updownarrow
↞	\twoheadleftarrow	↠	\twoheadrightarrow	⇕	\Updownarrow
↺	\circlearrowleft	↻	\circlearrowright	⇅	\updownarrows
↶	\curvearrowleft	↷	\curvearrowright	↑	\uparrow
↩	\hookleftarrow	↪	\hookrightarrow	⇑	\Uparrow
↽	\leftharpoondown	⇁	\rightharpoondown	↑↑	\upuparrows
↼	\leftharpoonup	⇀	\rightharpoonup	↟	\twoheaduparrow
⇇	\leftleftarrows	⇉	\rightrightarrows	↿	\upharpoonleft
↫	\looparrowleft	↬	\looparrowright	↾	\upharpoonright
↙	\swarrow	↘	\searrow	↓	\downarrow
↖	\nwarrow	↗	\nearrow	⇓	\Downarrow
↜	\leftsquigarrow	⇝	\leadsto, \rightsquigarrow	↓↓	\downdownarrows
⟺	$\iff$	↡	\twoheaddownarrow	⇂	\downharpoonleft
⟹	$\implies$			⇃	\downharpoonright

Punctuation

∵	\because	·	\cdot	·	\cdotp
⋯	\cdots	·	\centerdot	:	\colon
⋱	\ddots	…	\dots	.	\ldotp
…	\ldots	…	\textellipsis	∴	\therefore
⋮	\vdots	„	\quotedblbase	"	\quotedbl

Symbols mainly for mathematical use:

$\aleph$	\aleph	$\amalg$	\amalg	$\angle$	\angle
$\approx$	\approx	$\approxeq$	\approxeq	$\asymp$	\asymp
$\backsim$	\backsim	$\backslash$	\backslash	$\barwedge$	\barwedge
$\between$	\between	$\bigcap$	\bigcap	$\bigcup$	\bigcup
$\bigsqcup$	\bigsqcup	$\biguplus$	\biguplus	$\bigvee$	\bigvee
$\bigwedge$	\bigwedge	$\bot$	\bot	$\bowtie$	\bowtie
$\Bumpeq$	\Bumpeq	$\cap$	$\cap$	$\Cap$	\Cap
$\circeq$	\circeq	$\complement$	\complement	$\cong$	\cong
$\coprod$	\coprod	$\cup$	\cup	$\Cup$	\Cup
$\curlyeqprec$	\curlyeqprec	$\curlyeqsucc$	\curlyeqsucc	$\curlyvee$	\curlyvee
$\curlywedge$	\curlywedge	$\dashv$	$\dashv$	$\dagger$	\dagger, dag
$\ddagger$	\ddagger, ddag	$\diamondsuit$	\diamondsuit	$\div$	\div
$\divideontimes$	\divideontimes	$\doteq$	\doteq	$\doteqdot$	\doteqdot
$\dotplus$	\dotplus	ℓ	\ell	$\emptyset$	\emptyset
$\eqcirc$	\eqcirc	$\eqslantgtr$	\eqslantgtr	$\eqslantless$	\eqslantless
$\equiv$	\equiv	$\eth$	\eth	$\exists$	\exists
$\exists!$	\exists!	$\fallingdotseq$	\fallingdotseq	$\flat$	\flat
$\forall$	\forall	$\frown$	\frown	$\geq$	\geq, \ge
$\geqslant$	\geqslant	$\gg$	\gg	$\ggg$	\ggg
$\gnapprox$	\gnapprox	$\gneqq$	\gneqq	$\gnsim$	\gnsim
$\gtrapprox$	\gtrapprox	$\gtrdot$	\gtrdot	$\gtreqless$	\gtreqless
$\gtreqqless$	\gtreqqless	$\gtrless$	\gtrless	$\gtrsim$	\gtrsim
$\hbar$	\hbar	$\heartsuit$	\heartsuit	$\hslash$	\hslash
$\iiiint$	\iiiint	$\Im$	\Im	$\imath$	\imath
$\in$	$\in$	∞	\infty	$\int$	$\int$
$\intercal$	\intercal	$\jmath$	\jmath	$\land$	\land
$\leftthreetimes$	\leftthreetimes	$\leq$	\leq, \le	$\leqq$	\leqq
$\leqslant$	\leqslant	$\lessapprox$	\lessapprox	$\lessdot$	\lessdot
$\lesseqgtr$	\lesseqgtr	$\lesseqqgtr$	\lesseqqgtr	$\lessgtr$	\lessgtr
$\lesssim$	\lesssim	$\ll$	\ll	$\lll$	\lll
$\lnapprox$	\lnapprox	$\lneq$	\lneq	$\lneqq$	\lneqq
$\lnsim$	\lnsim	$\lor$	\lor	$\ltimes$	\ltimes
$\measuredangle$	\measuredangle	$\models$	\models	$\mp$	\mp
$\multimap$	\multimap	$\nVDash$	\nVDash	∇	\nabla
$\natural$	\natural	$\ncong$	\ncong	$\neq$	\neq
$\neg$	\neg o \lnot	$\nexists$	\nexists	$\ngeq$	\ngeq
$\ngtr$	\ngtr	$\ni$	\ni	$\nleq$	\nleq
$\nless$	\nless	$\nmid$	\nmid	$\not\approx$	\not\approx
$\not\equiv$	\not\equiv	$\not\sim$	\not\sim	$\not\simeq$	\not\simeq
$\notin$	\notin	$\nparallel$	\nparallel	$\nprec$	\nprec
$\nsim$	\nsim	$\nsubseteq$	\nsubseteq	$\nsucc$	\nsucc
$\nsupseteq$	\nsupseteq	$\ntriangleleft$	\ntriangleleft	$\ntrianglelefteq$	\ntrianglelefteq
$\ntriangleright$	\ntriangleright	$\ntrianglerighteq$	\ntrianglerighteq	$\nvdash$	\nvdash
$\nvDash$	\nvDash	$\oint$	$\oint$	$\parallel$	\parallel
∂	\partial	$\perp$	\perp	$‰$	\perthousand
$\pm$	\pm	$\prec$	\prec	$\preccurlyeq$	\preccurlyeq
$\preceq$	\preceq	$\precnsim$	\precnsim	$\precsim$	\precsim
$'$	\prime	$\prod$	\prod	$\propto$	\propto
$\Re$	\Re	$\rightthreetimes$	\rightthreetimes	$\risingdotseq$	\risingdotseq

⋊	\rtimes	♯	\sharp	∼	\sim
≃	\simeq	⌣	\smile	∢	\sphericalangle
⊓	\sqcap	⊔	\sqcup	⊏	\sqsubset
⊑	$\sqsubseteq$	⊐	\sqsupset	⊒	\sqsupseteq
⊂	\subset	⋐	\Subset	⊆	\subseteq
⊊	\subsetneq	≻	\succ	≽	\succcurlyeq
≿	\succeq	⋩	\succnsim	≿	\succsim
∑	\sum	⊃	$\supset$	⋑	\Supset
⊇	\supseteq	⊋	\supsetneq	√	\surd
±	\textpm	×	\times	⊤	\top
△	\triangle	⊎	\uplus	⊨	\vDash
⊩	\Vdash	∨	\vee o \lor	⩒	\veebar
‖	$\Vert$	⊪	\Vvdash	∧	\wedge o \land
℘	\wp	≀	\wr		

Other symbols

¶	\P	§	\S	℃	\celsius
✓	\checkmark	℧	\mho	Ω	\ohm
°	\textdegree	№	\textnumero	␣	\textvisiblespace

Appendix C
Index of commands

The commands discussed in this introduction are listed in this index. Some are simply mentioned, almost in passing, in which case the page that appears in the index indicates where they are mentioned. But other commands are the subject of some more detailed explanation. In this case, only the place where the detailed explanation begins is listed in the index, although the command may be cited in other places in the introduction as well.

Not included in the index:

- The `\stopSomething` that close a construction previously opened with `\startSomething`, unless the text says something special about the `\stop` command, or is treated at a different location than the one where the corresponding `\start` command is.
- Commands aimed at generating symbols, all found in Appendix B.
- In the case of commands that generate a diacritic or letter, and that have an upper case and a lower case version, to generate the upper case or lower case letter respectively, only the lower case version is included.

***