1

There are a bunch of books about the TEX typesetting language, but none of those books deal with aspects about the real purpose of TEX. The issue is typesetting, and this is much more important than any syntactical or semantical featurers that a language like TEX offers. Typesetting – apparent from the existence of the word – must be something different to word-processing and simple document printing. Typesetting is more than lining up characters in a row, either in proportional or in nonproportional fonts. Typesetting involves more than selecting fonts for flashy appearance. Typesetting is an art.

I'd like to drop the introductory subject entirely and dive directly into the question: How to use this bewildering kind of a typesetting language called TEX.

Fonts

Let's talk about fonts first. Fonts in general are divided in three different classes.

	sans serif	serif	fancy
also called			
	grotesque	antiqua	bastard

There are two very simple rules about fonts. Never mix fonts from different classes! Use fewer fonts! The font classes again are divided into subclasses.

normal
slanted
italic
condensed
light
bold

These subclasses are often used in arbitrary combinations (e.g. bold-condensed-slant). There are undocumented punch lines of wisdom about the combination of fonts from a single class. I recommend not to use slanted fonts for titles or headlines. In normal text highlighting is better done with slant than with bold.

Then there is the question of font selection. Which font, where? Here are some rules of thumb.

Long text for poetry or paperback-books should be written in serif fonts. Documents with lines longer than 60 characters ought to be typeset in serif fonts to avoid tiresome reading. Lines should always be as short as possible; this is the reason for multiple columns in newspapers. Reading is much easier the shorter the lines.

Short lines and short pieces of text may be typeset in sans serif fonts. Sans serif fonts are usually hard to read. Therefore they are usually sized larger, but this will be discussed when we turn to the issue of typesetting measurements.

Here are some typical fonts for typesetting:

serif	sans serif	fancy
Garamond	Helvetica	computer modern fibonaccy funny
Palatino	Optima	Corinna
Times-Roman	Univers	Baskerville
computer modern roman		

This is – in a few words – the way to select fonts, acquiring taste for good fonts takes a lot of practice. It's like composing music. But it is good advice to keep in mind the basic rules that grew with the art of typesetting during the centuries.

Here again the rules:

1.) Don't use many fonts

2.) the longer the document, the fewer fonts

3.) Documents with long lines need serif fonts
Serif fonts are easy to read.

4.) Slant is better than bold highlighting.

5.) Headlines not in slant

6.) Never slant capitals

7.) When normal text is serif, Headlines ought to be serif, too.

How to select fonts in TEX? TEX is a programming language like any other. Any programming language has data types, like real, integer, double, char and so on. The same is true with TEX. TEX data types refer to typesetting elements. Here are some:

font
box
tokens
numbers
dimensions
files

This is the essence and will be subject for the next pages: How to use the different data types. The issue now is fonts. For those who are familiar with programming languages will easily understand no problem to understand the way fonts are used in TEX. A symbolic name is introduced (for your specific document), to replace the real font. Here an example:

```
\font\a=Garamond at 10pt
```

This enables the font Garamond, which again may be referred to as `\a`. This leads to a substantial discussion about commands in TEX.

TEX commands

As any computer language TEX has keywords and a specific syntax and semantics how to use them. The problem with TEX is that keywords and normal text have to be seperated during the typesetting process as fast as possible. Therefore each keyword simply commences with a specific escape character, which may be any character in the pattern of your choice. For ease of use (and in order to avoid deep discussions) we adopt the character, that the inventor of TEX, Donald E.Knuth chose when he designed TEX. This character is \. So any sequence of characters preceded by \is considered to be a TEX keyword. This does not yet explain how TEX figures out when the end of a keyword is found. There is an easy rule. All characters ranging from A-Z and a-z are considered elements of a keyword, all other characters, special symbols and numbers are used as keyword delimiters. This may easily be changed, but leads to an academic discussion about characters and semantics. Refer to chapter 8 "The characters you type" in the TEXbook.

Now we know `\font` selects a font if the keyword `\font` is followed be a symbolic name, here `\a` and = and finally the name for the real font here Garamond' and additional keywords follows to set the size of the font, here 10pt. Eventually the symbolic font name represents a TEXkeyword, which may easily be detetected in the TEX-parser due to the preceeding

\escape character. The specific kind of the macro isdetermined by the font-declaration statement. The keyword \a refers to something wich turns out to be a font and more specific the font Garamont.

TEX knows a few ways to select fontsizes.

```
\font\a=Garamond at 10pt
\font\b=cmr10 scaled\magstep2
\font\c=cmr10 scaled 1432
```

The first way is the professional way to pick a font. It selects the font Garamond in 10pt. But fonts are not necessarily selected this easy. On standard printers fonts like Garamond are not available at all. Even though you may easily find Times-Roman font on common laser-printers, you may not randomly size this kind of font to any desired size. The same is true for all the fonts coming with TEX. Those fonts come in discrete sizes in sort of quantum leaps and are down-loadable pixelfonts. Pixelfonts have to be generated for specific resolutions depending on the quality of your output device. Therefore there is no chance to manipulate their size them dynamically, instead there are files containing the pixel informations for all different kinds of magnifications. TEX Versions come with all computer modern fonts in the typical resolution of 300pt. They are stored in the so-called generic font format, a format produced by Metafont, the sister-program of TEX, that can produce fonts in any imaginable shape. This format allows compact storage of pixel fonts. Actually generic font files use significantly less space than the equivalent pixel files. The printer-driver will automatically produce the according download format for pixel-fonts for your device, so nobody has to care about this.

If you look at these kinds of fonts, the so called "computer modern font family", you'll notice that they come in different directories. These directories contain the different pixel files in different steps of magnification, e.g. m300 contains all fonts in 300 dots per inch resolution in the standard design size, that is magnification 0. Plain TEX provides some predefined macros to pick fonts in different sizes. For esthetical reasons magnification is done with a factor of 1.2. That is a font like cmr10, which originally was designed for 10pt is called cmr10.300gf, the same font now comes scaled 1.2 times bigger than the initial font, these are contained in the directory m360. The next directory is m432, which contains fonts 1.2 larger than the m360 fonts, and this is continued until m1290. We therefore obtain 10 different directories containing fonts in the resolution of 300 dots per inch in 10 different magnifications.

And here is a little table for the easy selection of font sizes in TEX:

magstep	scaled	pt	gf
0	1000	10	300
half	1100	11	330
1	1200	12	360
2	1440	14.4	432
3	1728	17.3	518
4	2073	20.1	622
5	2488	24.9	746
-	2986	30	896
-	3583	35.8	1075
-	4300	43	1290

TEX supplies magsteps up to 5,this is why the table ends abruptly in the column magstep. There are apparently three ways to pick a font. Here some examples:

```
\font\a=cmr10 at 17.3pt
\font\a=cmr10 scaled 1734
\font\a=cmr10 scaled 1728
\font\a=cmr10 scaled1728
```

All pick the same fontsize, even the scaled 1734 will give you the font cmr10 scaled 1728, because fonts come in discrete sizes and the driver will pick the closest solution, but the spacing will not fit correctly, because TEX itself calculates the typesetting with an imaginary font 1734. This may be used to stretch characters apart like (s t r e t c h). The table also contains point sizes, which are applicable just for fonts designed for 10pt, like cmr10, cmss10 or the like. For fonts like cmr6 or cmsy8 this list can not be valid because those fonts obviously are designed for different basic sizes, here 6 and 8pt.

Sizes and measurements

In typesetting an awful lot of sizes and units have to be expressed in highest accuracy up to 1/1000 of an inch. Accuracy is the name of the game, when it comes to real typesetting. A „real“ typesetter will not be convinced by a

neat screen display, by an approximate „looks quite nice“ job. The real test for a typesetting solution is how it comes on paper, and nothing but the exact values will be accepted by the pro.

TEX knows a bunch of different kinds of measurement units. There are:

`pt`	pica points
`pc`	pica
`in`	inch
`bp`	bigpoint
`cm`	centimeter
`mm`	millimeter
`dd`	didot points
`cc`	cicero
`sp`	scaled points

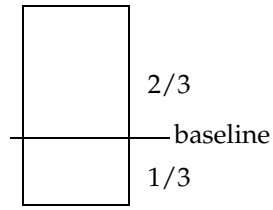
What are all these measurement units for? `pt` is used to determine small sizes for fonts. All fonts are measured in pt, like in our example above

tt\font\a=Times-Roman at 10pt

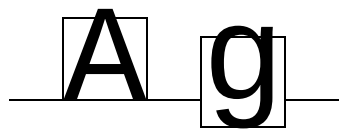
.

10pt is a very common size for standard book-printing. Even smaller fonts are used for special editions. The paperback edition of „War and Peace“ for instance, occasionally has something like 8pt sized letters and even things like 8.4pt are found. This book for example uses 10pt. Fonts bigger than that are usually used for headlines, and book-headers and titles. Professional typesetters usually don't like big fonts. Using big fonts indeed is not the job of a typesetter, rather the job of a layouter. People who do commercials and advertising deal with bigger fonts and combine single characters, which again is an art of its own. So let's talk about classic documents with small characters. Readable books are composed of `8pt` to `10pt` characters. Footnotes go down to `7pt` and `6pt` depending on the size of the fonts in the rest of the document. Shifted numbers (sunscripts & superscripts) are as small as `3pt` to `4pt`.

The bottomline of this discussion about the unit `pt` is: `pt` is used to determine the size of fonts. But what is the size of a font? This question is of basic importance for the use of a typesetting system. A font consists of different characters. A **g** definitely has a different size than the capital **A** of the same font. But we still talk about a font of lets say size 10pt. The explanation can easily be derived from the following diagram:



Each character sits in an imaginary box, which is 10pt high. 2/3 of the box is the character height and 1/3 is the character depth and each character has its own individual width. Back to our example. The **A** and bold g would look like this:



The widths of those characters are definitely different, but they both sit in a box, that is 10pt high. For sure you already noticed, that this is a simplification and abstraction. In reality each character has it's own width and depth. The depth of a capital **A** is zero and the height of the **g** is smaller than that of the **A**. But in general this abstraction helps a lot, it tells you that the deepest character of size 10pt is 1/3 by 10pt deep and the tallest character of size 10pt is 2/3 by 10pt high. This is a very important clue for the future discussion.

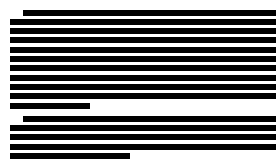
Let's continue with the next measurement units. For non-typesetters units like *pica* and *cicero* are probably items from outer space and in fact we can divide the units TEX can handle into two parts. The first part contains measurement units, which are historical to the art of typesetting (point, pica, didot point and cicero). The other part (inch, centimeter and millimeter) are the units we are supposed to be familiar with. Doing real typesetting means getting acquainted with real typesetting measurement units. We were already talking about fonts. When we chose a font of size 10pt it would mean to choose a font of 0.13837 inch. This apparently is less convenient to get along with, than 10pt. No wonder, that typesetters chose their own way to express the sizes they had to deal with in different units, due to the small dimensions. Therefore it is a good idea to remain in the world of typesetting units when a useful result is expected for the according document.

Almost everything can and sometimes has to be given in exact sizes for any document. Lets just list all interesting parameters for dimensioning your output, that are available in TEX:

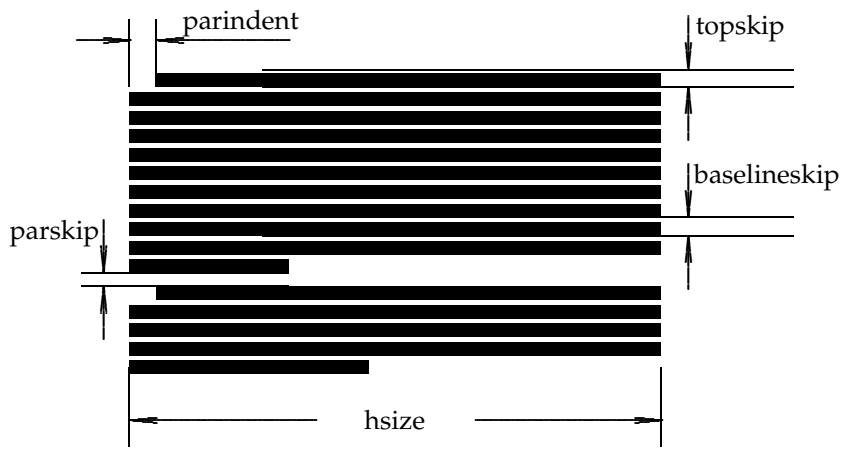
```
hsize          maxdepth
vsize          topskip
baselineskip   splitmaxdepth
parindent      boxmaxdepth
hangindent     hoffset
parshape       voffset
```

Apparently TEX provides all imaginable dimensions for complete control over your document.

Very rarely you need all of them, but there are important ones you really should keep in mind. Let's consider a very simple document of just two paragraphs and a very simple layout like this:



This layout really is as simple as can be, but still, it involves parameters to get it done right the way we want it. First rule to approach a typesetting system: never rely on default settings. You are the artist and you determine the appearance of your document. Let's look at our layout as a technical drawing:



A sum-total of 6 dimensions have to be specified to get this layout the way we want it. Let's take typical sizes:

```
\parindent=1cc
\topskip=.6666\fontsize    %   2/3 fontsize
\parskip=3dd
\baselineskip=11dd
\vsize=16cc
\hsize=12cc
\font\a=Times-Roman at 10pt
```

This exactly is the parameter list you have to specify to get the desired result. You may try this right away. When your TEX is installed, take an arbitrary text containing pure ASCII text, put those parameters on top, `\bye` at the end and squeeze it through TEX. The result ought to be just, what we've explained above. Take your favorite text-editor e.g. `vi ftry.tex` and type the following in your file `ftry.tex` (first try in tex)

```
\font\a=Times-Roman at 10pt
\parinent=1cc
\topskip=.6666pt
\parskip=3dd
\baselineskip=11dd
\vsize=16cc
\hsize=12cc
..... Text .....
\bye
```

Save your file, exit your editor and type:

```
ptex ftry
```

TEX is started and produces a single page indicating this with a `[1]` on your screen. Two new files will now exist. `Ftry.log` and `ftry.dvi`. The `.log` file contains all possible errors. There should be none, unless you made a mistake in typing, or `.......Text......` contains things like `\`, `&`, `%`. Avoid those characters for this first try. When you successfully filtered your file `ftry` through TEX, you now may print it on your Post-Script laser-printer with:

```
psdvi ftry |lp
```

And after a short while your page is supposed to appear on your laser-printer. No doubt many things can go wrong, if so your TEX installation is not correct please refer to your installation manual.

You now have supposedly produced your first TEXdocument. This should encourage further steps to learn more about TEX.

Important parameters for typesetting

In our previous example we produced a page with a few paragraphs and just one single font, nevertheless the result looks pretty neat. This is what typesetting is all about, not a flashy appearance and a whole lot of fonts, but readable and pleasing text output. To stress this point even further I'd like to seduce to an additional, but most interesting experiment, with right the same text we just used to produce our first sheet of paper with. As you most likely noticed TEX produced justified output, and does hyphenation. How come? Well TEX has it's own hyphenation clockwork built in, to cut words into pieces. Hyphenation is a problem of it's own and we'll not bother with it. Just keep in mind that TEX can do it and better than other programs can. The important point is how TEX does the justification. The answer to this question is of significant impact to the use of TEX and all typesetting problems you will try to tackle with it. TEX does no simple line breaking by merely adding the width of each character in words and trying to fit it into a line with a specific length, instead it collects possible line breakpoints and picks the optimal breakpoints in order to achieve an optimal visual result of a paragraph under specific parameterized conditions. That sounds very academic and theoretical. Therefore I'd rather like to attract your special attention to the following examples:

Return to the keyboard and edit this `ftry.tex` file and add the following line just on top of`........ Text ........`

```
\spaceskip=2.78pt plus 4pt minus .8pt
```

Run your file through TEX and produce a new output. The result is different and you probably notice, that it looks nicer, more professional. What we added with this line is a new rule for interword spacing, called `\spaceskip` in TeX. The parameter states the rule used as standard interword spacing 2.78pt up to 6.78pt and as low as 2pt. The appearance of your document changes stunningly by just changing this single parameter `\spaceskip`. TEX wouldn't be TEX if there were no parameters to take into account that are closely connected to a parameter like spaceskip. Indeed there is an additional parameter called `\xspaceskip`. This parameter gives the word

spacing after specific characters i.e. after .,!?, or the like. Rarely it is good style to make a difference between those two cases – normally interword spacing and spacing after punctuation are equal – but especially in the USA you can meet this kind of bad habits. I recommend that whenever you change spaceskip simply change `\xspaceskip` to the same value. So in our example the complete rule would need the additional statement:

```
\xspaceskip=\spaceskip
```

And now you should try to play with these values for spaceskip and watch the different results. You will notice that, the less you allow a plus and minus, the more often TEX will tell something about overfull and underfull boxes. This will give you an idea of how sensitive these values will react on different `\hsize`. The narrower your document gets, the more plus value has to be allowed to achieve justification. However the fixed avarage value of `\spaceskip` has to be chosen carefully, too, in order to get a pleasing justification of your paragraph.

Those who already peered into the TEXbook might notice the similarity of the introduction but probably will miss a discussion about `\tolerance` and `\hbadness`. This is on purpose, because the effect of those values is significantly different to what we are talking about right now, but later on, we are going to talk about those values as well.

It is important to play around with these values in order to develop a good feeling for the right choice of those values. You probably noticed during your experiments, that a single value of `\spaceskip` i.e. something like `\spaceskip=2.73pt` is valid as well. This value will result in a raggedright output, however not in the way one would possibly like it, which is due to an awful lot of over- and underfull boxes.

Now that you know how `\spaceskip` influences the behaviour of TEX line breaking algorithm, we might as well try to do a good ragged right output. Even though this might appear as an absolutely trivial task, in the eye of a professional typesetter it turns out to be one of the most difficult task. So what is good ragged right in the eye of a pro?

Ragged right should avoid hyphenation. The interword spacing is fix, and to be real correct the lines should be broken long, short, long .... . As with any problem a computer should solve, careful analysis of the given task is more important, than a full featured programming language. To fulfill the condition of alternating lines length is somewhat more difficult and will be discussed later, but the other rules can instantaneously be fulfilled with primitive means in TEX.

Avoid hyphenation. There is a specific parameter for hyphenation called

\tolerance, which will be explained in detail a few pages later. \tolerance has to be set to 10000 in order to avoid hyphenation. So much for now. The fixed interword spacing is easily accomplished with a single value for \spaceskip. Finally we have to provide a real soft margin in order to justify the fixed word spaces with a stretchable lineend. This is right the way how TEX thinks about this problem, therefore there is nothing like a right or left margin, but instead there is a \leftskip and a \rightskip, respectively. Hence you may add the following lines to your text for ragged right margins:

```
\spaceskip = 2.78pt
\xspaceskip=\spaceskip
\pretolerance=10000
\rightskip=0pt plus 3cc minus 0pt
```

A rightskip of 0pt plus 3cc means each individual line may be 3cc shorter than \hsize. If a negative value greater 0pt is given, the lines may consequently stick out \hsize by the given amount.

You may certainly figure out how a centered text with ragged left and right margins will be done.

Right we just mentioned the parameters \leftskip. When

```
\leftskip=0pt plus 3cc
```

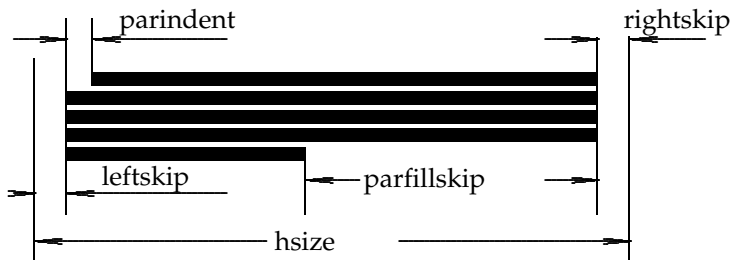
is added to your file it ought to produce left and right centered single lines of each paragraph, but actually it doesn't. There is an additional parameter that controls the behaviour of a paragraph: \parfillskip.

\parfillskip has a default value (mind the default values) in order to fill the last line of each paragraph. This is not what we want in centered lines. So we simply set:

```
\parfillskip=0pt
\parindent=0pt
```

You may play with these parameters and remove your left and right skip commands and notice interesting effects.

Let's summarize the new parameters in a small diagram:



We get good control over the appearance of a paragraph with these parameters, but the appearance of a paragraph comes in a larger variation than we can handle with these few parameters. Let's take examples from everyday life. Numbered paragraphs, paragraphs with big initial capitals, shaped paragraphs. You may easily notice the interdependence, the mutual influence between all mentioned parameters. But let's introduce a few more parameters to handle typical applications.

Consider a paragraph like the following:

This is something very common. Paragraphs preceded with dots, numbers, or a.) b.) c.) or the like. Sometimes a complete word precedes a paragraph. Let's take a simple one, and the according TEX commands and discuss the result later.

1.) 

The first line of this paragraph starts with 1.) and a space. How can we manage to have the following line indented exactly by the length of 1.) and a space?. This leads to a very neat and fundamental feature of TEX.

14 Horizontal boxes

We have already been talking about fonts and that each character of a font is surrounded by an imaginary box. If it were possible to put those boxes into a new box it should be possible to know the resulting size of the final box, This is actually feasible with TEX and is a specific data type like `\font` is. This data type is called `\box`. We can declare a symbolic name for a box. i.e. `\indtxt` by simply saying:

```
\newbox\indtxt
```

and now we can say

```
\setbox\indtxt=\hbox{1.) }
```

The box `\indtxt` now contains 1.) and must have the resulting width of all four characters. We can check this easily by saying:

```
\showthe\wd\indtext
```

and TEX will print the width of indtxt on the screen, saying something like

```
> 14.4467pt
```

This demonstates an important feature. TEX provides means to trace almost any internal result of its own calculations. Accordingly you can easily find out the height of `\box\indtxt` and depth of this box by saying:

```
\showthe\ht\indtxt
\showthe\dp\indtxt
```

The box has the height of the tallest character it contains and a depth of the deepest character it contains. This gives the opportunity even to find the depth height and width of any character combination and even of a single character.

As an example:

```
\setbox\indtxt=\hbox{g}
\showthe\wd\indtxt
\showthe\ht\indtxt
\showthe\dp\indtxt
```

will give the exact values of the surrounding box of the character "g". But TEX not only can show you the values but also you may use those values to assign parameters i.e.:

```
\parindent=\wd\indtxt
```

This may even be used with parameters like spaceskip

```
\spaceskip=\wd0
```

And this is useful in the example lines from a few pages before.

```
\spaceskip=2.78pt plus 3pt minus .78pt
```

may as well be stated as:

```
\setbox0=\hbox{i}
\spaceskip=\wd0 plus 1.3\wd0 minus .3\wd0
```

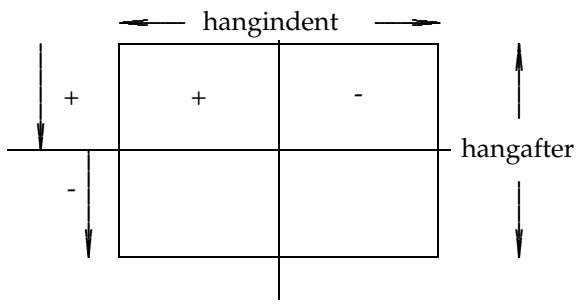
This is what the typesetter would actually say and do. The command says: the spacing between words is the width of an "i" in the chosen font, the space may stretch to 130% and shrink to 30%. This fits much more in the pattern of thinking of a professional typesetter. And the result shows it.

As you will have noticed I did not declare a `\newbox` but instead used a number. This may always be done to suit the user's laziness, yet it has the disadvantage that you have to remember all the box numbers and you cannot refer to boxes by symbolic names.

Back to our indentation problem. When we declare a box with:

```
\setbox\indbox=\hbox{1.) }
```

we would know exactly how deep the second and consecutive lines have to be indented – this is `\wd\indtxt` – as long as we can find a parameter that allows us to indent the second and consecutive lines. There is such a parameter, called `\hangindent`. `\hangindent` offers more than just a second line and the rest of the lines of a paragraph to be indented. Together with the parameter `\hangafter` you may easily shape your paragraph in a great variety of forms. A little diagram shows more:

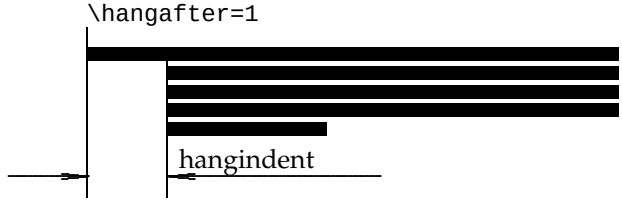


As long as the value of hangindent is positive the paragraph will be indented at the left and, when it gets negative it will be indented at the right. Whereas the value of hangafter says from which line indentation

should start (with a positive value) or at which line it should end, as soon as it is negative.

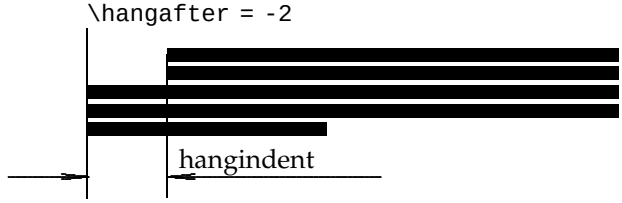
Here some examples:

```
\hangindent=2cc
\hangafter=1
```




```
\hangindent=-2cc
\hangafter=1
```


```
\hangindent = 2cc
\hangafter = -2
```




```
\hangindent=-2cc
\hangafter=-2
```

These are all the possibilities. Let's transfer this knowledge to our everyday life problems of a paragraph that started with 1.)

```
\parindent=0pt
\newbox\indtxt
\setbox\indtxt=\hbox{1.) }
\hangindent=\wd\indtext
\hangafter=1
\box\indtxt....... text ..........
```

This may sound difficult, but in reality it isn't. What's that `\box\indent` all about, why can't we simply start our text in the following way:

1.) ..... text .....

Very easy; the result would be something like this:

1.) The beginning of the first line would not be aligned with the following lines. Why is that so? Interword spaces may be stretched or shrunken and this is against our intention.

The beginning of the first line would not be aligned with the following lines. Why is that so? Interword spaces may be stretched or shrunken and this is against our intention. The first space between 1.) and the text has to be fix and may not serve for any line justification. The space in the `\box\indtxt` contains a space, which has right it's standard average value. This is obvious, because hboxes extend exactly to their natural width, that accumulates by adding the width of all characters the box contains, no stretching or shrinking. This gives us an opportunity to get a closer look to the property of boxes.

A box like:

```
\hbox{abcdef}
```

contains but the character "abcdef". In order to construct a box with more complex content, there must be some additional commands to handle this, e.g. a box that contains this:

a. b.

Let's Suppose we exactly know the distance between a. and b. we could use the command `\hskip`

```
\hbox{a.\hskip1in b.}
```

supposing the distance between a. and b. is an inch. In typesetting we usually encounter a different problem: a. and b. have to be placed in such away, that they span over 1 inch. This can easily be done with:

```
\hbox to 1in{a.\hfill b.}
```

Sometimes more complicated solutions are needed, in particular boxes when boxes overlap.

```
\hbox{\box\A \hskip-8mm\box\b}
\hbox to2cm{\box\A \hss\box\b}
```

Now we've learned a lot about horizontal boxes and it's time to return to practical applications.

First let's summarize the keywords we've learned

`\hbox`	defines a horizontal box
`\hfill`	fills a box
`\hss`	fills a box, if necessary overruns box borders

With this we can easily center single words in a line of length `\hsize`

```
\hbox to \hsize{\hfill ....Text....\hfill}
```

This sounds easy, but what happens when `....Text .....` gets longer than`\hsize`? TEX will utter something about an overfull box. There are several interesting questions, that arise from this. When text spreads to more than `\hsize`, what could it have been, what our initial intention was. There are a few possibilities, that represent what we might have liked to see:

1. 2. 3. 4. 5.

The first case may simply be treated as stated above.

```
\hbox to \hsize{\hfill .....Text.........\hfill}
```

The second one would not work with these commands, instead you have to use:

```
\hbox to\hsize{\hss....Text...\hss}
```

You probably remember the discussion a few pages ago. So you may say:

```
\spaceskip=2.78pt
\xspaceskip=\spaceskip
\pretolerance=1000
\rightskip=0pt plus 5cc
......Text......
```

In order to get the result for our third case. After `....Text....` you'd've been forced to reset all parameters to their original settings, which is easy

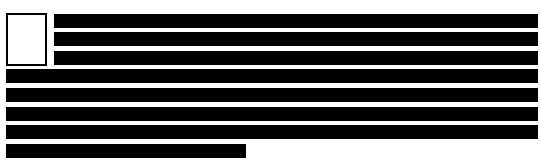
supposing you really know them. It probably would require to debug the default settings first by saying:

```
\showthe\spaceskip
\showthe\pretolerance
\showthe\rightskip
```

in order to know the exact values for these values, so you may reassign them after `....Text.....` Many high-level programming languages don't need that, they are block-structured and stack-oriented and so is TEX. The only thing you need is some kind of bracketing. In TEX the symbols for brackets may be chosen at will, but the default setting appears to be practical and useful and it reminds of other programming languages. Block begin is {and block end is }. The easiest solution to reset your parameters is embracing your block with brackets.

```
{\spaceskip=2.78pt
\xspaceskip=\spaceskip
\pretolerance=1000
\rightskip=0pt plus 5cc
.....Text.....}
```

We also showed how to do centered text with several lines and numbered or itemized paragraphs. So we presumably can handle all these kind of layout elements for a document's page. But what about something popular like this:



Here we can't use any of those parameters we've learned so far.

Let the first character be a "C" in `Univers-CondensedBold`. The height of this character apparently is `3\baselineskip` of the capital height of our normal font. This is also a good example how to handle sizes in TEX.

```
\dimension\Cheight
```

defines a new variable for a dimension.

```
\Cheight=3\baselineskip
```

assigns the value of 3 baselineskips to the variable `\Cheight`.

```
\setbox0=\hbox{AFGC}
\advance\Cheight by \ht0
```

`\box0` contains capitals in the normal font. So `\box0` has to have the height of the capital letters of this font, more precisely, the height of the highest of those 4 characters AFGC, which ever this is. We advance `\Cheight` by the height of the capitals in the standard font. `\Cheight` now contains the desired height for the capital height of our paragraph initial, which is a C. But still it would not be correct to say:

```
\font\b=Univers-CondensedBold at \Cheight
```

The resulting characters, in particular our C, would turn out to be way too small to cover the first 3 lines. Remember our discussion about fonts. So in fact we have to choose the font for our C in the following way:

```
\font\b=Univers-CondensedBold at 1.5\Cheight
```

The capital height is 2/3 of the fontsize. Consequently the fontsize has to be chosen 1.5 times bigger in order to get a C that is exactly `\Cheight` high.

This is one way to do it. But let's talk about real fancy character styles in a font, where 2/3 is not the general height of any capital character. isn't there a chance adjust the size of a character C that will fulfil our need for a character being `3\baselineskip` high.

There sure is a way.

```
\newdimen\size
\size=10pt
\loop
\font\b=Univers-CondensedBold at\size
\setbox1=\hbox{\b C}
\advance\size by 1pt
\ifdim\ht1 < \Cheight\repeat
```

This is a way to iterate the desired size to an accuracy of 1pt, which is fairly appropriate for this kind of application. But we ought not to exaggerate the accuracy, because TEX will select a new font every time the loop is iterated, which finally will fill the entire internal TEX memory with font descriptors. But TEX not only provides loops like any other computer language, but also means to do calculus to a certain extend. There is multiply, divide and advance, where advancing something by a negative value correspond to substraction. Thus no need for a special function for this. But there are many different types of data, which are not necessarily compatible.

```
\newcount\a
\a=1
\advance\a by 1
```

is definitely correct

```
\newcount\a
\newcount\b
\a=1
\b=10
\advance\a by\b
```

still is right.

```
\newdimen\a
\newcount\b
\a=1cm
\b=1
\advance\a by \b
```

will not work due to the incompatibility of types. You certainly may not mix counts and dimensions, but you may easily say:

```
\advance\a by\b mm
```

At the end you may mix all kinds of data-types, if you keep track of the according units.

Calculating is very important for typesetting. Almost all sizes have some sort of interrelationship. Therefore calculus is important. There are a few ways to calculate with TEX. The commands

```
\multiply
\divide
\advance
```

just do integer calculation, so they may not be used for any calculation where relations are expressed by fractions. Things are more difficult in those cases. Have a look at the macro-package PICTEX, to see how a floating point divide and multiply may be programmed in TEX.

It took us a long way just to pick the right size for our paragraph initial but, we still have no paragraph with it. Now we want to produce a paragraph with the following characteristics:

The first line is `\hsize` long with the initial sticking down two lines. So

the second and third line is shorter by the width of the initial character C and all other lines are `\hsize` long again. This is something you can't easily obtain with commands like `\hangindent` or `\parindent`, because they simply do not provide enough flexibility to control more than one line or consecutive lines with different behaviour. (For those who know TEX and are protesting, I simply say there are always 142 and one way to solve the problem).

Now let's first contemplate how to manipulate the initial C so that it will behave the way stated above.

Vboxes

Characters come in boxes . The size of the box is determined by the height of the containig character and it width. So our C initially looks like this:

A paragraph simply written with this kind of C would come like this:

which is not what we want. The C has to be changed into a flat box capital height high of our normal font. Let the normal font be:

```
\font\a=Times-Roman at 10pt
```

Remember how to obtain the height of capital characters in our font `\a`:

```
\setbox0=\hbox{ABCTL}
```

The height of `\box0` became the height of the capitals ABCTL. There is no need to put all capital characters in `\box0` but just to make sure I picked a private collection; you may as well just use one single A. We now would like to put our C in a box that is `\ht0` high and C stick out at the bottom. Here is how to do.

```
\vbox to\ht0{\hbox{C}\vss}
```

There is a command called \vbox giving control over vertical boxes of any height, in our case a box with \ht0. Vboxes have a very special behaviour. They usually extend to the width of the surrounding box, in our case \hsize, unless they contain nothing but a \hbox. So this:

```
\vbox to\ht0{C \vss}
```

is a C in a box \hsize long, that is the complete length of the first line. Containing nothing but a \hbox does not necessarily mean a limitation to a single \hbox. Complex constructs like:

```
\vbox{\hbox{...\vbox{\hbox{\hbox.....}}}}
```

are undoubtedly valid, in fact boxes may be nested as deep as your imagination and concentration reaches, but to make the width of a box be controlled by what it contains, the first inner element must be a \hbox and nothing but a \hbox please note things like:

```
\vbox{ \hbox{...
```

will extend the \vbox immediately to the size of the surrounding box, usually \hsize. Worse: it will create a single line with a space and a error message that says something about an underfull box and a second line with the contents of the \hbox.

If we now put our C in a \vbox at height \ht0, then C sits in a box much higher than \ht0 so, the \vbox will be overfull because the lower part of our C sticks out at the bottom. Therefore typing something like:

```
\vbox to\ht0{\hbox{C}}
```

would result in an error message saying Overfull vbox badness 10000. In order to avoid this, there are similar positioning commands as for an \hbox. All of them now start with v

```
\vfill
\vss
\vskip
```

A \vfill is a stretching element extending only in the positive direction. It is useful to center objects in a vbox, i.e.

```
\vbox to 20\baselineskip{\vfill .... Text....\vfill}
```

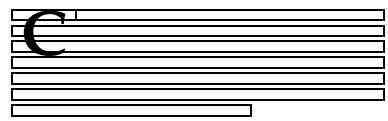
or in combination with \vskip it may be used to position material at an exact place.

```
\vskip3cm ....Text....\vfill
```

or from the bottom

```
\vfill ....\vskip3cm
```

We will use this later to construct page layouts and you will see that working with boxes is a very important and frequent job. Back to our C in the box. The `\vbox to\ht0{\hbox{C}\vss}` is right the kind of character we need to construct our paragraph. Using this character will result in the following output:

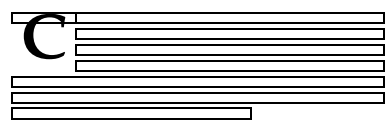


This is much closer to what we intended, but how we get control over the next lines? Here is a new command that does exactly this: `\parshape`. With parshape you may assign each line of a paragraph a specific length and indent, here the values for our case:

```
\length=\hsize
\advance\length by -\Cwidth
\parshape 0pt\hsize \Cwidth\length \Cwidth\length 0pt\hsize
```

There are apparently 4 pairs of numbers giving for each line the individual indent and the according line length. The first parameter of parshape says how many lines will have individual sizes and then the sizes follow, first the indent and then the line length. This is why we had to calculate the length first by substracting `\Cwidth`. The careful reader might have noticed that just four lines are given instead of 7, which is the total number of lines of our sample paragraph. `\parshape` features the property to maintain the last value for the rest of the lines in a paragraph.

Here it is at last, our paragraph with a big initial C:



And this is the set of commands that does it:

```
\font\a=TimesRoman at 10pt
\setbox0=\hbox{\a ABCTL}
\Cheight=\ht1
\advance\Cheight by 3\baselineskip
\font\b=Univers-CondensedBold at 1.5\Cheight
\setbox1=\hbox{b C}
\vbox to\ht0{\hbox{\b C}\vss}
\length=\hsize
\advance\hsize by -\wd1
\parshape 4 0pt\hsize \wd1\length \wd1\length 0pt\hsize
......Text....
```

Sounds awful, doesn't it. Well for ease of use TEX has an additional feature, that makes life easier when we need special effects like those more often. It would be tedious to do all the writing over and over again just to get this kind of effect on the plot. Therefore we can simply assign a name to all this and use it as a macro.

Macros

With this feature our clumsy something above would turn into a handy little command of our own choice, like this:

```
\def\I{\setbox0=\hbox{\a ABCTL}
     \Cheight=\ht1
     \advance\Cheight by 3\baselineskip
     \font\b=Univers-Condensed-Bold at 1.5\Cheight
     \setbox1=\hbox{b C}
     \vbox to\ht0{\hbox{\b C}\vss}
     \length=\hsize
     \advance\hsize by -\wd1
     \parshape 4 0pt\hsize \wd1\length \wd1\length 0pt\hsize}
```

But macros can do a lot more. Suppose you need this initial letter game

for different applications, and especially for different initials. Macros can have parameters, so that something like this may be written:

```
\I C .....Text.....
```

We simply change our macro into the following

```
\def\I#1{...
```

The parameter `#1` will be replaced by the according character, in our case a C. You may have more than one parameter. More precisely apparently 9 because there is just one character available to specify the parameters number. Unfortunately you may have no parameters with two or more digits, but I will show solutions for this in special applications. In our case, for instance, we supposedly would like to give the number of lines the initial character should cover. So we can write:

```
\I2 C...Text... or \I1 C...Text...
```

But this would require a mechanism to alter the parshape command by other commands.

So lets have a closer look at macros. A brief glance in the TEXbook in the chapter macros, reveals a bunch of different commands to define macros:

```
\def
\edef
\xdef
\gdef
```

With the help of those and some additional mechanism, we can achieve the most tricky results to define and manipulate macros. Our goal is the automatic construction of a parshape command depending on a given number of lines that we want to indent for the initial character. In case of just one line our parshape command has to look like this:

```
\parshape 4 0pt\hsize \Cwidth\length \Cwidth\length 0pt\hsize
```

The more lines we want, the more often the part `\Cwidth\length` has to be repeated. In a classic programming language this would look like this:

```
A$="\parshape 4"
for i=1 to N Where n is the desired number of lines
 A$=A$ + "\Cheight\hsize"
next i
A$=A$ + "0pt\hsize"
```

TEX provides all necessary commands like `if`, `then`, `else` and even

the command `loop`. So lets start to translate our Basic Program into TEX commands:

```
\def\A{}
\loop\ifnum\i < #1
  \edef\A {\A \Cwidth\length}
  \advance\i by 1\repeat
\edef\A {\parshape\i 0pt\hsize\A 0pt\hsize}
\A....Text....
```

This is it, the command `\edef` makes it possible. This command simply expands any macro in a new definition, before completing this definition. `\A` is empty at the beginning, walking through the loop it initially yields `\Cwidth\length` then `\Cwidth\length \Cwidth\length` and so on and we finally get as many `\Cwidth\length` commands as we want. When the loop is completed, we may give our `\parshape` the finishing touch with the same trick. But we have to remember that we added two additional line parameters and have to increase the number of lines we want to control by 2. The complete macro to produce initials of arbitrary size looks like this:

```
\def\I#1#2{\setbox0=\hbox{\a ABCTL}
        \Cheight=\ht1
        \advance\Cheight by #1\baselineskip
        \font\b=Univers-Condensed-Bold at 1.5\Cheight
        \setbox1=\hbox{b C}
        \vbox to\ht0{\hbox{\b C}\vss}
        \length=\hsize
        \advance\hsize by -\wd1
        \def\A{}
        \loop\ifnum\i < #1
         \edef\A{\A \Cwidth\length}
         \advance\i by 1\repeat
        \edef\A{\parshape\i 0pt\hsize\A 0pt\hsize\A}}
```

At this point we almost know all the different mechanisms to manipulate a single paragraph. What we don't know is how to create perfect lines and perfect line breaking. We were already talking about hyphenation and the importance of interword spacing, but there is more than this. Take a simple example we are all familiar with: paperback books, also called pocket books. These books have a very simple layout, just plain text of 40 to 65

characters per line, one single font and nothing special, you might think. But the typesetting of those books is governed by very subtle rules to improve readability.

Breaking Lines

The idea is to get paragraphs with just a few hyphens, the fewer the better, and if possible no hyphenation in the line before the last line in a paragraph. In addition, if there are hyphenations necessary, which is hard to avoid, no two consecutive lines should be hyphenated in order to avoid a right paragraph border, that looks like a comb. All this can be controlled with TEX and this is where we go deep into the theory and practice of how TEX does linebreaking and why. To commence with this of subject, we first should become familiar with the parameters and their meaning, that influence linebreaking. Then we will talk about the TEX process and interaction of those parameters.

```
\tolerance\pretolerance
\hyphenpenalty\exhyphenpenalty\linepenalty
\adjdemerits\doublehyphendemerits\finalhyphendemerits
```

Only an example can help to a closer understanding of what happens in TEX with these parameters. Lets take a text-material like the following in 10pt Times-Roman, typeset to 10cm width:

```
When will we meet again in thunder lightning or in rain.
```

The quality of a breakpoint depends on all of the above parameters. TEX assigns different brakpoint a badness value in the following way:

The width of the characters are added and constantly compared to the desired length, usually `\hsize`. The closer we get to `\hsize`, the more the line may be justified by the ”stretchability“ of the interword spaces. But because spaces may be stretched and shrunken, TEX will find several, usually two breakpoints for one line. In our example these breakpoints would be:

```
When will we meet again in
```
or
```
When will we meet again in thun-
```

There are two possible ways to typeset our phrase. The first one takes more stretching, whereas the second one is denser and already uses shrinking. The first line therefore has a higher value for badness than the second one, consequently TEX would pick the second breakpoint. This kind of automated line-breaking probably might not be our choice. We have all kinds of parameters to influence TEX's decision in our direction of thinking. The badness of each breakpoint is compared to the values of `\pretolerance` and `\tolerance`. This is because TEX first does line-breaking without hyphenation. For the first pass the resulting breakpoints are compared in their values of badness with pretolerance. If pretolerance is big enough, it will always cover even the worst breakpoints. In our example the breakpoint after "in" has a badness of 2400. Any pretolerance higher than this would favour that breakpoint. A value smaller than 2400 apparently will disallow this breakpoint. This ensues a value of -1 to disallow all breakpoints found in the first pass, where no hypenation is applied. This does not necessarily mean that a pretolerance of -1 results in all lines being hyphenated, but it says the first pass trying it without hyphenation will always fail. Whereas a value of 10000 will always succeed, because no breakpoint will ever have a badness greater than 10000. Remember our ragged right paragraph we had as example a few pages ago. Here is the reason why it works. When it comes to the second pass, all the words in a paragraph are cut into pieces through the hyphenation algorithm. And the process of assigning badness to breakpoints recommences. This time breakpoints with hyphens are added to the list to those without hyphens. The badness of each breakpoint is compared to a value called `\tolerance`, and again, if `\tolerance` is high even lines with high badness are considered acceptable. The higher the `\tolerance`, the more lines may be stretched to fit into the desired justification margins, this results in an awful looking paragraph. In good typesetting an exact value for the interword stretchability and shrinkability and a low value of `\tolerance` would be desirable. Something like:

```
\tolerance=10
\xspaceskip=\wd0 plus 1.3\wd0 minus .3\wd0
\xspaceskip=\spaceskip
```

is a very reasonable choice. But this still would not avoid esthetically bad breakpoints like consecutive hyphenated lines or the second-last line to be hyphenated. Additional parameters serve to tackle this problem.

```
\hyphenpenalty\exhyphenpenalty\penalty
\linepenalty
```

With these penalties you may assign badness values to each resulting breakpoint. These values are taken into account to rate the quality of a breakpoint. So hyphenation may as well be avoided by assigning 10000 to `\hyphenpenalty`. Due to performance reasons this is not the recommended way to avoid hyphenated lines. Any smaller value, though, is the only feasible method to get paragraphs less hyphenated. For paperback-books a value of 5000 is reasonable. With the general parameter penalty you may assign any position in your paragraph a specific breakability or unbreakability. Consider the following example:

```
a distance of about 10 m
```

It definitely will lead to an unpleasant appearance if just by coincidence the linebreak occurs between tt10 and `m`. So you may use plain TEX's tie ~. which turns out to be

```
\def~{\penalty10000\}
```

As you may see, the following breakpoint before a space is prohibited due to the largest possible penalty just before the space. Please note: don't try it this way:

```
\def~{\\penalty10000}
```

This still would incidentally result in a line broken just between `10` and `m`. Breakpoints are calculated before spaces and not after them. Therefore this command is entirely obsolete. But still the solution for our above example is not very professional. Expressions like 10 m or 100 C have a fixed spacing between the value and its unit, so something like this is much better:

```
10\kern3pt m
```

Kerns are not considered valid breakpoints unless they are followed by some stretchable item like a space again.

Sometimes it is desirable not to avoid breakpoints, but instead to force them explicitly. In this case negative penalty values help. Values like

```
\penalty=-500
```

tells TEX that this point is acceptable in case of doubt, but it not necessarily results in a line that is broken at this place, when TEX finds an even better place on it's way through the paragraph. If you prefer a breakpoint like the following:

```
a distance of
about 10 m
```

you may suggest this to TEX by typing:

```
a distance of\penalty=-500 about 10 m
```

The more negative the penalty value gets the more this breakpoint will be considered to be a good one and if you say `\penalty=-10000` the line will be broken at this point by brute force. This is very helpful to control individual lines in a compound paragraph. There exist 3 typical forced line breaks in the typesetting world:

1.) line break with justification

2.) left aligned line

3.) right aligned line

The first case is simply `\penalty-10000` also defined as `\break` in plain TEX. The second case requires, that the line is filled with empty space, which may easily be done with a `\hfill\penalty-10000`. The last version is not possible with TEX, because we get no clue about the typesetting process and linebreaking before it is actually done. Because we have no real knowledge about how lines finally come out, when we are in the process of typing them, therefore we know nothing about the end of the previous line. As a consequence two lines have to be broken by hand to get the desired result. First a justified line preceding our right aligned line and the right aligned line itself.

```
.....Text....\break
\hfill....Text....\break
```

This may sound reasonable, but it will not work. TEX removes all empty material after breakpoints, so we loose the effect of `\hfill`, which forces us to put some empty horizontal material at the beginning of the line. Here is how it works:

```
\hbox{}\hfill....Text....\break
```

The hbox is empty, but it is not discardable as a lone `\hfill` would be.

What we derive from this is that TEX is not built for single-line breaking.

We are still talking about parameters influencing line break. There are a few left unmentioned, in particular the demerits.

These values may be used to influence the special cases I mentioned twice, e.g. for the production of paperback-books. The parameter `\finalhyphendemerit` forces or prohibits the second last line to be hyphenated. `\doublehyphendemerit` serves to avoid or force two consecutive lines to be hyphenated and the tricky parameter `\adjdemerits` deals with line compatibility.

This sounds much scarier than it is in practice. Each line gets a rating whether it's loose tight or very loose. Two consecutive lines may now in

accordance with `\adjdemerit` be loose followed by tight or tight followed by very loose.

The demerits differ from penalties not only in name. The values for the demerits have to be chosen much higher than penalty values to achieve the desired effect. If we want to avoid hyphenation in the second last line, a finalhyphendemerit of 100000000 would do it for sure, whereas a value of 10000 would not guarantee the same effect. This is because demerits have to be chosen as the squares of penalties.

Here are some practical examples to demonstrate the effect of our parameters in the real world applications.

The idea is to get paragraphs with just a few hyphens, the fewer the better, and if possible no hyphenation in the line before the last line in a paragraph.

```
tolerance=100
pretolerance=0
linepenalty=5000
hyphenpenalty=0
doublehyphendemerits=0
adjdemerits=0
finalhyphendemerits=0
```

The idea is to get paragraphs with just a few hyphens, the fewer the better, and if possible no hyphenation in the line before the last line in a paragraph.

```
tolerance=100
pretolerance=0
linepenalty=0
hyphenpenalty=10000
doublehyphendemerits=0
adjdemerits=0
finalhyphendemerits=0
```

The idea is to get paragraphs with just a few hyphens, the fewer the better, and if possible no hyphenation in the line before the last line in a paragraph.

```
tolerance=100
pretolerance=10000
linepenalty=0
hyphenpenalty=0
doublehyphendemerits=0
adjdemerits=0
finalhyphendemerits=0
```

The idea is to get paragraphs with just a few hyphens, the fewer the better, and if possible no hyphenation in the line before the last line in a paragraph.

```
tolerance=100
pretolerance=-1
linepenalty=0
hyphenpenalty=0
doublehyphendemerits=0
adjdemerits=0
finalhyphendemerits=0
```

The idea is to get paragraphs with just a few hyphens, the fewer the better, and if possible no hyphenation in the line before the last line in a paragraph.

```
looseness=1
tolerance=1000
pretolerance=0
linepenalty=0
hyphenpenalty=0
doublehyphendemerits=0
adjdemerits=0
finalhyphendemerits=0
```

The idea is to get paragraphs with just a few hyphens, the fewer the better, and if possible no hyphenation in the line before the last line in a paragraph.

```
looseness=1
tolerance=100
pretolerance=0
linepenalty=0
hyphenpenalty=0
doublehyphendemerits=0
adjdemerits=0
finalhyphendemerits=0
```

The idea is to get paragraphs with just a few hyphens, the fewer the better, and if possible no hyphenation in the line before the last line in a paragraph.

```
looseness=1
tolerance=1000
pretolerance=0
linepenalty=0
hyphenpenalty=0
doublehyphendemerits=0
adjdemerits=0
finalhyphendemerits=
            100000000
```

The idea is to get paragraphs with just a few hyphens, the fewer the better, and if possible no hyphenation in the line before the last line in a paragraph.

```
looseness=1
tolerance=1000
pretolerance=0
linepenalty=0
hyphenpenalty=0
doublehyphendemerits=
                100000000
adjdemerits=0
finalhyphendemerits=0
```

The idea is to get paragraphs with just a few hyphens, the fewer the better, and if possible no hyphenation in the line before the last line in a paragraph.

```
looseness=1
tolerance=1000
pretolerance=0
linepenalty=0
hyphenpenalty=0
doublehyphendemerits=0
adjdemerits=100000000
finalhyphendemerits=0
```

The idea is to get paragraphs with just a few hyphens, the fewer the better, and if possible no hyphenation in the line before the last line in a paragraph.In addition, if there are hyphenations necessary, which is hard to avoid, no two consecutive lines should be hyphenated in order to avoid a left paragraph border, that looks like a comb. All this can be controlled with TEX and this is where we go deep into the theory and practice of how TEX does linebreaking and why. To commence with such a kind of subject, we first should become familiar of the parameters and their meaning, that influence linebreaking. Then we will talk about the TEX process and iteration of those parameters.

```
rightskip=0pt plus .3cc minus.3cc
parfillskip=0pt
spaceskip=2pt plus.1pt minus.1pt
tolerance=9900
pretolerance=-1
linepenalty=0
hyphenpenalty=0
doublehyphendemerits=0
adjdemerits=-100000000
finalhyphendemerits=0
```

The professional typesetter might have noticed, that something is missing that most typesetting systems provide for better hyphenation control, i.e. a mechanism to predetermine the length of the pieces a word might be broken into by the hyphenation process. This has been introduced in the 3.0 version of TEX.

We now have considered different examples for single paragraphs with the main objective of esthetical quality, but in some cases more than this required is required. In paperback-books you may never use vertical glues in order to construct a page of a desired length. But as an additional constraint pages may never lead to widowlines on the following page. The only reasonable solution is to typeset paragraphs one line longer than what they might naturally be typeset obeying the according parameters like penalties and tolerances. For this purpose TEX has a parameter called `\looseness`. The `\looseness=1` i.e. leads to a paragraph that is one line longer than its natural length, but only when `\tolerance` allows this. Unfortunately TEX has no parameter that would tell about the success of looseness. This results in very complicated actions to be undertaken in order to determine the paragraph, that will get one line longer with the smallest necessary tolerance.

The preceeding chapters demonstrated the difficulty of creating paragraphs and headlines of esthetical quality. This is very important for the understanding and use of any typesetting system, because in the real world of typesetting, producing output doesn't commence with the page

layout, footnotes and tricky multicolumn pages, but with the single lines itself. A typesetter always produces a proof or galley first. This galley was and still is cut into pieces and glued into the page layout by a different person who actually has a different profession. This person is taking care of mounting the page layout, and you wouldn't believe that still today, with all WYSIWIG and screen oriented typesetting masterpieces of software, it is very hard almost impossible, to beat a person doing a layout by hand with a computer. I would't recommend a tournament where TEX or any other typesetting system software tries to combat hand assembly. But TEX can do automatic mounting in a page layout, which we will discuss in the following chapters. In these cases TEX can do assembly work much much faster than anybody.

This is true for all kinds of material that may be standardized e.g. pocket books, monthly or weekly reviews and many others. This kind of literature comes in standard formats and as a matter of fact the editor and publishers standardize layouts and usually they stake the rules the layout should look like.

Producing a complete page of typeset material may be roughly divided into three steps:

1.) produce a galley

1.a let it proofread

1.b produce a corrected galley, repeat 1, 1.a, 1.b until satisfied

2.) cut the galley at the appropriate places

3.) glue the pieces into the predetermined layout.

What sounds easy, actually isn't. To the contrary the point of discussion will be: what are the appropriate places

Both examples show bad places to cut our sample galley into pieces. It sure is a bad idea to have the header of a chapter sitting on the bottom of our column or page and the rest of the chapter starting on top of the next page or column. This is also true for a first single line hanging onto the header, loosing the rest in the middle of nowhere. But when other breakpoints are better, i.e. two or more lines together with the header, we run into further trouble, because the pieces we cut from our galley incidentally tend to be too long for our page or they get too short. bottomline: choosing the right point to cut a galley is a very delicate task.

This reminds us of the problem of breaking paragraphs into lines and indeed it turns out to be a very similar problem. it just rotates the problem by `90`$^{\text{fl}}$ from horizontal to vertical. this is the way how TEX thinks about

the problem and we consequently find two modes in TEX, a vertical and a horizontal mode.

most of our parameters that we've learned for the horizontal mode will not be applicable for the vertical mode, yet one of them is `\penalty`. so we actually have a parameter in common for the horizontal and vertical mode. with this parameter it must be possible to control our galley-cutter. Here an example:

```
\Header
\penalty 10000
.....Text.....
```

This sounds simple. `\Header` now will not be split off the rest of the Text, but how to avoid the cut between the first two lines of `....Text.....`? Well, there are similar parameters for vertical breaks, as there are for horizontal breaks. Here a summary:

```
\clubpenalty
\widowpenalty
\brokenpenalty
\interlinepenalty
```

The `\clubpenalty` forces or prohibits the first line of a paragraph to be considered a reasonable breakpoint. The `\widowpenalty` does the same for the last line of a paragraph. The `\brokenpenalty` serves for lines that end with a hyphenated word. And `\interlinepenalty` is an overall penalty for all lines in a paragraph. We notice the same flaw in the TEX concept that appeared for breaking lines in a paragraph. We have no parameter to assign penalties to individual lines i.e. the third or fourth line in a paragraph, so that we can't control the size of the pieces a paragraph may be split into.

I already mentioned the effect that cut pieces of our galley tend not to fit into our page layout. There are two cases to be considered. Either we may justify lines by stretchable and shrinkable elements like interword spaces or we may not. This is true for vertical lists, too. Either we may put stretchable or shrinkable elements in the vertical list i.e. between paragraphs or we may not.

If stretching and shrinking is prohibited for linebreaking, TEX consequently produces lines of different length. If stretching and shrinking is prohibited verticaly for page-breaking, the result will be pages of different length, which usually is not quite what we want.

We remember the according parameter \spaceskip and \xspaceskip for horizontal material. For the vertical mode there is \parskip and the equivalent parameters to \hfill, \vfill and \hss, \vss, that we already mentioned for vboxes.

The problem of skips

In general we choose a font for our document first. The font size requires an appropriate baselineskip, which is very important for the appearance of a document's page. A typical font size of 10pt may be typeset with a baselineskip of 11pt, just to give an example. But there are other possible solutions. In typesetting we have three major different kinds of baselineskips.

1.) compressed

2.) normal

3.) wide.

Compressed is easy to choose. Take the font size in pt, choose the same value for baselineskip in didot. i.e.

```
\font\a=Times-Roman at 10pt\a
\baselineskip=10dd
```

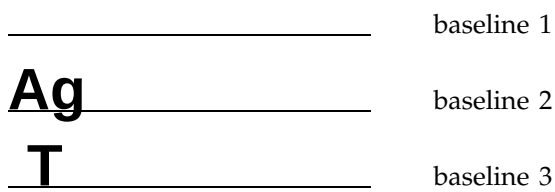
Characters will never touch each other on top and bottom, because didot are slightly bigger than points are. Compressed is used for footnotes, marginal notes, abstracts and special headlines.

Normal is 110% of the font size. A 10pt font would require a baselineskip of 11dd for the normal text-body. The book you hold in hand is configured this way. A 20pt font would require a 22dd baselineskip to conform to this rule. In the real world the rule is applicable for fonts from 8pt to 17pt. Bigger fonts need more spacing between lines, therefore the relationship between font-size and baselineskip is not linear for all font sizes.

Wide will therefore be used for large fonts or document pages with a sketchlike outfit with typewriter-font, i.e. Wide is chosen by multiplying the font size by 1.4 and using this value in didot as baselineskip.

Eventually the value for baselineskip is logically chosen bigger than the

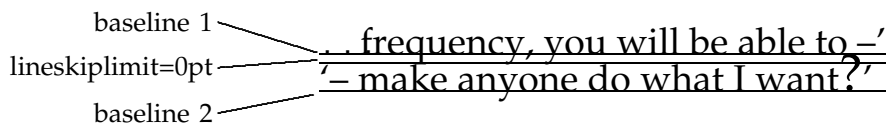
value for the font size. A normal capital letter will consequently always fit into the space between two baselineskips



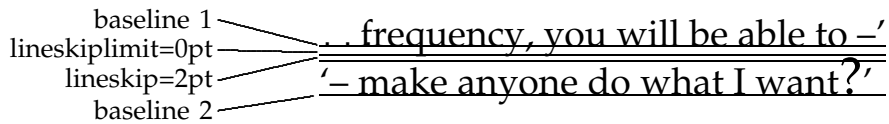
No collisions will ever occur even in compressed format. Remember: the entire character fits into a surrounding box of fontsize. 1/3 of this box is the height of the capital letters and 1/3 is the depth of special characters like g,j,p,q,y. But what happens if a line contains oversized characters or other oversized elements? To handle those kinds of problems TEX supplies two additional parameters for baselineskip

```
\lineskip
\lineskiplimit
```

Those two parameters control the behaviour of baselines in case they get too close together. Let's consider the following example:



The `lineskiplimit` represents an imaginary boundary beneath the standard baseline. A threshold trespassed by the height of any character like the question mark in the above example triggers TEX to undertake action in order to separate lines for minimal readability by the amount of lineskip. Suppose `lineskiplimit=2pt` and `lineskip=3pt`, `baselineskip=11dd`. The following would happen:



The value of `lineskip` will be the distance between the deepest character on baseline 1 and the highest character (here the question mark) on baseline 2.

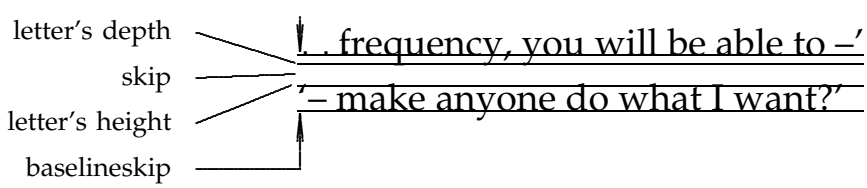
This effect is not what a professional typsetter would ask for, especially because `lineskip=0pt` would not deactivate this kind of action to be performed by TEX.

Usually something like this would never be accepted in the pro's world, because it never happens unless by mistake. In this case any default action triggered by default values is a disastrous result (mind the default values). The only way to get rid of this well-meant feature is to say somthing like

```
lineskip=0pt
lineskiplimit=-baselineskip
```

Now the oversized element must be at least two baselineskips high to trigger the automatic justification. The justification will then be `0pt`.

A deeper discussion about TEX vertical skips is the unavoidable consequence of the above discussion. The mechanism of lineskip already shows the way how TEX calculates interline spacing. TEX would always do the following: it gets the deepest character of the preceding line, takes the largest character in the next line and calculates a skip positioning the second line that sits exactly on a baseline of the first line. To get this straight, here a picture:



This is of significant importance for the understanding of how lines will be stacked up a pile. Getting rid of the baselineskip consequently is a delicate process. The statement

```
\baselineskip=0pt
```

would produce a total mess when the values of `lineskip` and `lineskiplimit` remain unconsidered. For instance,

```
\lineskip=0pt
\lineskiplimit=0pt
\baselineskip=0pt
```

would not result in

‘– make anyone do what I want?’ but ‘. . . frequency, you will be able to –’
‘– make anyone do what I want?’

`lineskip` is the skip between lines and not the skip where baseline will go. What you see here is the result of the following proceeding of TEX: The deepest character of line 1 and the highest character of line 2 will overlap due to the value of `baselineskip`, which is `0pt`. The `lineskiplimit=0pt` therefore will activate `lineskip`. This is always true, whenever a character of one line will overlap characters of the following line. `lineskip` is `0` consequently lines will be spaced by the deepest character and the highest characters, respectively. If overlapping lines are the intention, our parameters have to be something like the following

```
\lineskip=0pt % any value
\lineskiplimit=-2\baselinskip
\baselineskip=0pt
```

What happens is that due to the negative `lineskiplimit` no action will activate `lineskip`. Therefore the value of `lineskip` is of no influence in this case. Please remember the way TEX piles up lines. It always calculates the difference between the highest and deepest characters of two consecutive lines. In our case this will necessarily result in a negative value. We now have to make sure to avoid the appearance of lineskips. `lineskip` is triggered by `lineskiplimit`. As long as `lineskiplimit` has a bigger value than the skip that has been calculated by TEX in order to align baselines in a distance of `baselineskip=0pt`, `lineskip` will have no effect. This is done by

```
lineskiplimit=-2\baselineskip.
```

Consider a table with rules to seperate fields. Here something very simple

language	TEX, Prolog, Lisp
processor	mips, Trace, CM1
toy companies	MicroSoft, Tomy

This is as simple as can be and not a very practical case but it elucidates the proper proceeding in those cases

```
\def\L#1#2{\hbox{%
       \hbox to.6\hsize{\hfill#1}\kern6pt%
       \vrule height 8pt depth4pt width1pt\kern6pt%
       \hbox to .6\hsize{#2\hfill}}}
```

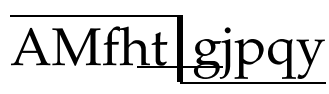
The first thing you will notice is that it will be a table but without the use of a special TEX command to deal with tables. TEX does not treat tables very different. The only difference to special table commands is that the field sizes for the largest table element may be calculated automatically.

```
\L{language}{\TeX, Prolog, Lisp}
\L{processor}{mips, Trace, CM1}
\L{toycompanys}{IBM, Tomy}
```

when these lines are preceeded by

```
\lineskip=0pt
\lineskiplimit=0pt
\baselineskip=0pt
```

Each line of our table contains a vrule

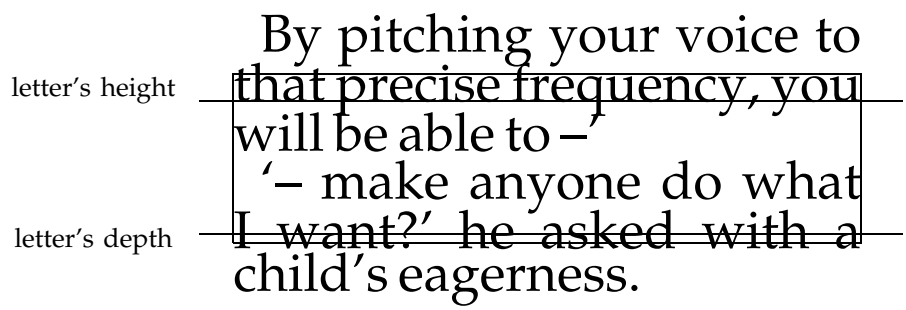


No character will be deeper than the vrule and no character will be higher, so the vrule is the dominating object governing the skipping process. The baselineskip is 0pt, consequently consecutive lines will always overlap. `lineskiplimit` is `0pt` and as a result lines that not even overlap but only

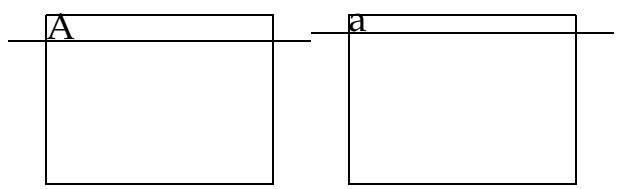
touch each other will trigger lineskip. This is true for every line. The lineskip is 0pt. The vrule will therefore sit exactly on top of each other.

This may now always be used when lineskipping is deactivated. This occurs in tables as shown, but also in footnotes.

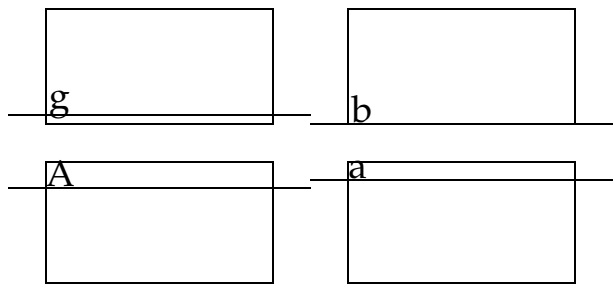
Remember the chapters about footnotes, where an insert-box collected the footnote material. What happens if a second paragraph of footnotes, that is a second footnote, comes along. The insert box is calculated the following way:



If topskip is omitted the first baseline in our insert box is determined by the highest character in the first line. Suppose the first line contains just lower case characters in the first case and capitals in the second case



There we have our first problem. There is one solution for the first paragraph in the `vbox` of our footnote, insert that is to choose `topskip` at least capital height high. This might help for the first paragraph in a footnote, but it is of no use for the second paragraph. We run into severe problems with four cases and their permutations.



In the first case the paragraph ends with a line containing characters with depth. In the second case the paragraph ends with a line without characters with depth. In the third case the next paragraph starts with a line containing capital characters, and finally in the last case the next paragraph starts with a line having no capitals. Proper alignment can not be assured under these conditions. In inserts (here footnotes) under compulsion paragraphs may not be aligned by the normal baeslineskip and parskip mechanismn applicable for the rest of the test. Because the footnote-test appears asycronous to the normal linebreaking algorithm. Consequently even with proper baselineskips in between footnote paragraph-lines, we won't have this for several of those paragraphs.

The idea to avoid all problems is very simple. The first and the last line of each paragraph has to contain an element at least capital height heigh of the chosen font and deep as the deepest character of the chosen font may get. This can be done with a so-called strut, which is a vrule very similar to the one used in our table above, but invisible. Our footnote paragraph now would look like this:

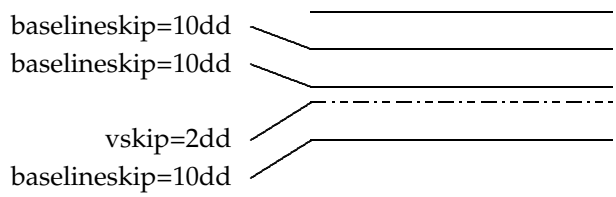
> By pitching your voice to that precise frequency, you will be able to –'
>
> '– make anyone do what I want?' he asked with a child's eagerness.

Having an invisible rule at the begining and end of the text guarantees proper results in all cases mentioned above. All those who always wanted to look behind the scenes of the bewildering footnote macro lined out in the TEXbook: here lies the key for the proper understanding.

The subtle features mentioned by Donald E. Knuth lie in the fact that in his

macro, each character is followed by an invisible rule. The recursive macro reads character by character and puts a vrule after each of it. This allows all imaginable manipulation to be applicable to those kinds of footnote insert, without running into the trouble of loosing the proper linespacing. In most of all cases the solution is overdone and very hard to read and to program. I therefore recommend the simple solution, lined out above.

There is nothing more to say about skips beside the general skip called `vskip`. `vskip` is simple to use and to understand. A `vskip` is added to `baselineskip`, when there is



a baseline. `vskips` are thrown away whenever a page break occurs. Take this book for example. Each chapter headline has a

```
\goodbreak\vskip3\baselineskip
```

before and

```
\nobreak\vskip2\baselineskip
```

after each headline. A pagebreak occurs preferably before the headline starts and is forbidden between headlines and the beginning of the chapter's text.

When the pagebreak really occurs at the beginning of a chapter it would not look nice if it started with 3.5 lines empty space. TEX knows this, and always discards obsolete skips. Therefore you may never start a page with

```
\vskip 10\baselineskip
MY ARTWORK
```

The ARTWORK would sit right on top of the page, because the skip will be discarded. If you need something like this, say

```
\hrule height0pt depth0pt
     \vskip3\baselineskip
```

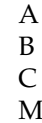
a rule, even the invisible one, may not be discarded. The only disadvantage of this method is the special behavour of `hrule`. If you say:

```
\hbox{abcde}        \hbox{abcdefg}
\hrule       or   \hrule
```

you will get abcde abcdefg

Note the difference! TEX will not calculate lineskips for hrules! This is no reason to worry. A hrule still is a prefect means to make vskips undiscardable. The following example shows, that the two expressions shown below are perfectly equivalent and represent 4 lines. This can easily be verified with the third diagram showing the corresponding size for four normal lines.

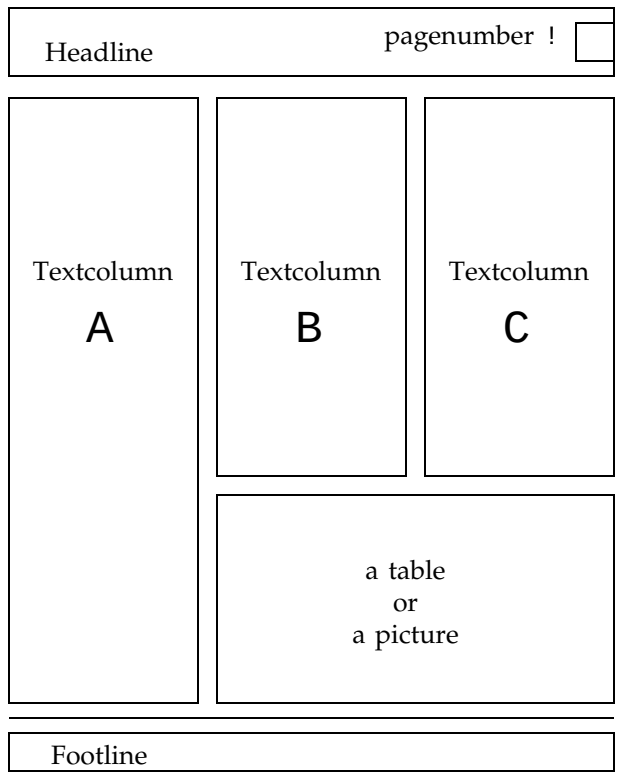
```
\hrule                \
\vskip3\baselineskip  and   \vskip3\baselineskip
My Artwork              My Artwork
```

My Artwork My Artwork My Artwork

Constructing pages

We learned about the mechanism how to cut the galley into appropriate pieces, now it's time to put pages together. The first step to do so is to analyse carefully the construction of your layout. Let's take an example



We can always easily construct the according assembly directives for TEX without any text in dealing with boxes that we might as well define right in the first place. Here we have

```
\newbox\columnA
\newbox\columnB
\newbox\columnC
```

Suppose the boxes A to C already contain typeset material. Then we can

simply start the construction of our page layout. Always start in horizontal direction with the smallest element from inside to outside.

```
\hbox{\box\columB\hskip6pt\vrule\hskip6pt\columnC}
```

We put these two columns in a vertical box together with the box `\tablebox`:

```
\vbox to\clength{%
   \hbox{\box\columB\hskip6pt\vrule\hskip6pt\columnC}%
   \hbox\tablebox}
```

I put % signs after each line, to avoid unsolicitated spaces in our `\vbox`. Column B and C are completed and represent a counterpart to column A, we can proceed with.

```
\hbox{\box\columnA\hskip6pt\vrule\hskip6pt%
\vbox to\clength{%
  \hbox{\box\columB\hskip6pt\vrule\hskip6pt\columnC}%
  \hbox\tablebox}
```

You probably have noticed that there is no need to give the length for the vrule. The `\vrule` extends to the length of the surrounding `\vbox`. The page layout is almost completed; just the header and footer are missing.

```
\vbox{\hbox to\pagewidth{Headline\hfill\the\count0}
   \vskip3mm
   \vbox to\clength{%
   \hbox{\box\columnb\hskip6pt\vrule\hskip6pt\columnc}%
   \hbox\tablebox}
\vskip2mm
\hrule
\hbox to\pagewidth{footline\hfill}}
```

you should carefully study this kind of page construction to get a full flavour of the power of this kind of solution. If the boxes `\columnA` through `C` actually already contain material you may simply say `\shipout` in front of the complete construction and your first page will come out. Unfortunately this is not the case unless your galley has been carefully cut already beforehand into the boxes `\columnA` through `C`.

Here now comes the interaction between your galley and the assembly of your page. Whenever TEX finds a good breakpoint in your vertical list it will immediately stop and execute a specific list of commands collected under the keyword `\output`. So we may say

```
\output={\vbox{\hbox to\pagewidth{textlayout\hfill\the\count0}
      \vskip3mm
      \vbox to\clength{%
     \hbox{\box\columnb\hskip6pt\vrule\hskip6pt\columnc}%
     \hbox\tablebox}
  \vskip2mm
  \hrule
  \hbox to\pagewidth{ME-magazine\hfill may 1990}}
```

Well, but this can't work, because the material, we collected is not put into the boxes columnA through C and how could TEX possibly know, that they have to be put there. Each time TEX figures a good break point for our galley it will activate the commands in the output routine. For one single page it will be our job to collect the material from the galley in the appropriate boxes, until we may start assembly work on our complete page.

The galley itself is a box, called box255 in TeX. This means TEX offers the galley cut at a certain point according our directives in box255. Nothing is easier than unpacking box255 and moving the material from box255 into a box of our choice. For example like this:

```
\setbox\columnA=\box255
```

This is a good idea, but in specific cases the results will be unexpected. Usually, also in this case, we expect a column to have a specific length. Due to many influences this is very unlikely to happen. TEX cuts columns either in the correct length or shorter, when it goes to the output routine. Consequently we usually have to deal with a column that is shorter than the desired column. If the column may be stretched vertically, then there is no problem, but the column still comes too short and how do we adjust the length now?

```
\setbox\columnA=\vbox to\Clength{\unvbox255}
```

Here is the way to accomplish our job. `\unvbox` actually unpacks the context of a vertical box and it may now be stretched again to the right size, which is done by saying `\vbox to\Clength`. This is the standard way to do it, and you should keep it in mind. If our `\box255` contains no stretchable material, we are stuck and have to use the simple way.

This is how our \output should look like:

```
\output={\if A\C
      \setbox\columnA=\vbox to\Clength{\unvbox255}
      \global\let\C=B\global\vsize=.7\Clength
      \else\if B\C
      \setbox\columnB=\vbox to.7\Clength{\unvbox255}
      \global\let\C=C
      \else\if C\C
      \setbox\columnC=\vbox to.7\Clength{\unvbox255}
      \global\let\C=A\global\vsize=\Clength
        .
        .
      assemple as above
        .
        .
      \fi\fi\fi}
```

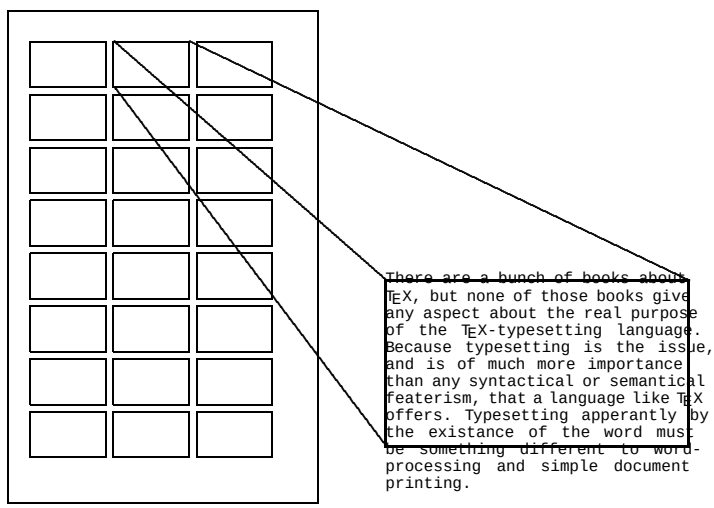
This output routine does the following. When TEX finds a suitable breakpoint it jumps to the output-routine and \box255 is unpacked and filled into \box\columnA. The Variable \C is set to B in order to activate the \else\if B\C path for the next time TEX returns with material to the output routine. For this event we prepare \vsize, which is smaller than \vsize for the first column. TeX then returns to work and continues to collect further vertical material, until it reaches \vsize again. Now \box255 will be packed into \box\columnB. Here we set variable \C to C. Column C having the same size as B spares us to readjust \vsize again. TEX returns for a collection of data and finally reenters the output routine again. This time the \columnC will be packed. The next pages are to be considered to have the same layout, we therefore readjust \vsize and set variable \C to A again. Now assembly begins, the page is shipped out and the process is repeated over and over again. Apparently this kind of layout making will not necessarily do everything we need and want, when the layout changes it will fail, and worse, what happens when the text ends? How should the layout look in this case and what happens in case TEX returns to the output routine with the rest of the galley.

We notice TEX's mechanism to build and assemble complete pages may not be used easily. Many things have to be considered and taken into account.

If the layout of pages is repeated over and over again, this kind of

handmade solution might sound reasonable enough, but it will be impossible to tackle when the layout changes entirely from page to page, as it does in a monthly or worse weekly gazette.

Instead a closer look to the real structure of a magazine page gets mandatory. Pages don't come in arbitrary layouts with pictures randomly distributed over the page. Pages usually have their specific kind of layout with recognizable rules.



The page is divided by a grid, that represents a whole bunch of different golden section, that may be filled with text and picture material. For ease of use the grid rectangles align with the baseline of the chosen font on the bottom and with the height of the capitals on top. Graphics, pictures and the columns may now be put into the shown grid. The result is a very versatile, but unified, layout for a page. The following pages are based on the same layout.

These pages are strictly organized and we may easily construct a perfect output routine, that is fed by general directives about the size and position of our pictures. Let's simply organize our raster in a matrix 3 by 8, giving the size and position of the picture in the following way

```
\pic(2,1;2,3)
```

determines exactly the appearance of our page.

This defines a picture, starting at element 2,1 and extending over two columns being 3 elements deep.

The height of a text element is defined by the simple rule:

```
n *(x * baselineskip + Capital-height + gap)
```

where gap is

```
2 * baselineskip - Capital height
```

giving the following final result:

```
n(x+2)baselineskip
```

Consequently our picture has the height:

```
3(9+2) baselineskip = 33 baselineskip
```

and a width of

```
(elwidth + gap) * 2
```

The lengths of our text-columns are perfectly determined:

height of column A = 8 (9 + 2)
B = 5 (9 + 2)
C = 5 (9 + 2)

With this we may build an output routine, that – in accordance with our picture definition – will automatically produce the textual columns.

The definition of our three pages may be done in the following way.

```
\page(\pic(2,1;2,3))
\page(\pic(1,1;2,2),
     \pic(6,2;2,3))
\page(\pic(4,1;2,2))
```

The three pages are defined as simple as can be, but the result depends on a lot of rules and involves a lot great deal of knowledge about good typesetting.

This is a perfect example for the quality of a typesetting language, where lots of typesetting rules may be hidden in a program. No mouse-hazzle, no approximation to absolute values, but quality output based on the experience and the rules of typesetters.

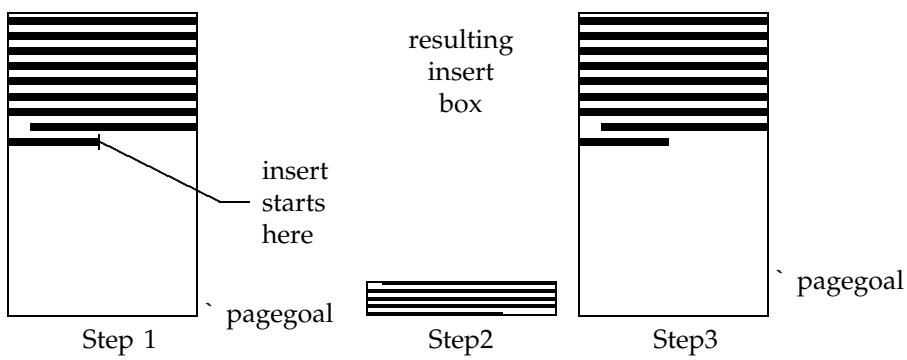
There are a lot more similar applications. One will be mentioned later again, the supposedly trivial paperback-books. This subject will show additional constraints emerging from classic typesetting rules.

When it comes to page layout there are still additional features you might expect in a typesetting system. The most popular, but – from the typesetters point of view – the most unpleasing layout element are footnotes. Therefore major rule in typesetting: avoid footnotes!

Footnotes

The structure of TEX allows to handle footnotes to any imaginable extend. By abstraction footnotes may be considered as inserts to the text body. TEX by it's design nature assumes footnotes to be inserted right at the spot in the text, where they will refer to.

TEX separates footnote and text-body and according to the macros undertakes the desired actions. This actually is as easy as it sounds. When TEX works on a text, it breaks lines in the way mentioned above. Lines are piled and built in a continuously growing block up to the point where the block gets as high as the pagesize (`\vsize`). At this time TEX jumps into the output routine. There are apparently two ways to jump into the output routine, either the block of lines shorter than `\vsize` or the pile of lines longer than vsize. In both cases the expected page or column height may only be adjusted by stretching the gap between paragraphs. But there are also applications disallowing any justification at all. When footnotes are taken into account, the process is perfectly equivalent. Lines are broken and piled but during line assembly a footnote occurs. What happens is that the pagesize to be reached by textbody is taken back by the appropriate amount of footnote height. You can picture yourself the expectable events, here some graphical illustrations.



This is the most common case, it has no unpleasing side effects, and is typical for the normal use of footnotes.

Unfortunately in some specific faculties (namely law) people tend to use footnotes remarkably immodestly. The documents significance is

underpinned by an unreasonable ratio of footnote to text, but books are produced in this kind of layout, so we might as well face the problem. A single footnote in a document of this kind may corrupt our good belief that the piled bodytext and the footnotes will necessarily fit on a single page.

This may result in two different decisions. The page may be typeset hopelessly overfull or the footnotes are broken the same way pages are broken.

This would result in a very interesting recursive behaviour of TEX. First pages are split then footnotes are spilt with the same rules, then footnotes in footnotes may be split.

This actually is not what happens. TEX restricts these kinds of events to one recursion level. That is just pages with footnotes, not footnotes in footnotes will be handled. This is reasonable, because in practice this never takes place.

As there is one additional sublevel for page breaking, consequently there are additional parameters for that kind of sublevel.

```
\splittopskip
\splitmaxdepth
\floatingpenalty
\insertpenalty
```

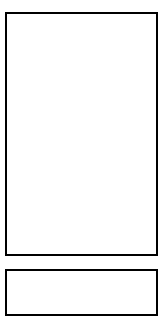
Now let's turn to a real example to get all mentioned circumstances straight.

```
\newinsert\a
\count\a=1000
\skip\a=24dd
......... Text ............
.........................
.........................
\insert\a{.....text.......}
.........................
..........
```

The `\newinsert` command will result in four new data-elements: a `\box\a`, where the inserted text finally goes. A `\count`, which is the magnification factor for page-breaking. A `\dimen` giving the maximum insertion size per

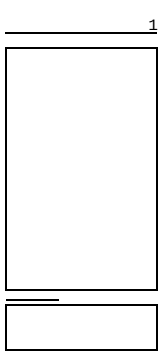
page and a \skip, which is extra space to be allocated on the page for insertion.

When the insertion occurs, a box with the material to be inserted will be constructed. Suppose it will turn out to be 36pt high. The pagegoal will be diminished by 36pt and, in addition, by the value of the dimen skip. The pagegoal consequently is smaller and the piling process for lines is interrupted earlier, resulting in a page much shorter than pages without insertion. TEX therefore enters the output routine with this shorter page and the assembly process in the output routine may now build the entire page consisting of the footnote-box and the box with the rest of the text. Here is a simple exercise:




```
\output={\vbox to\vsize{\unvbox255
                        \vskip\skip\a
                        \box\a}}
```

We can easily mount some flashy accessories around this:




```
\output={\vbox{\hbox to\hsize{\hfill\the\count0}
               \hrule
               \vskip8dd
               \unvbox255
               \vskip\skip\a
               \hrule width4cc
               \box\s}}
```

Looks like a very nice page that we can use immediately for a nice book

with footnotes. Note the construction `\unvbox255`. We know the textbody and the footnote-text together with the skip will not very likely exactly stretch to pagesize.

Therefore there are three solutions.

1. The textbody is stretched to the appropriate size, as in our example.

2. The the fixed skip is replaced by a vfill resulting in a stretch or shrink to accommodate vsize or

3. the footnote-box is unboxed and may stretch. Definitely we can do combinations of all three above. In the eye of a typesetter the third solution is no reasonable approach, above all because in most cases foonotes will not come as several paragraphs. Stretching will therefore be impossible. Even with several paragraphs this would result in an unpleasing picture, with the bottom paragraph stretched apart and the top textbody strictly fixed between paragraphs. The second solution, though sounds realistic and may preferably be used, when equal spacing between paragraphs in the main textbody is desired. With a footnoteline a solution according to this would probably look like this:

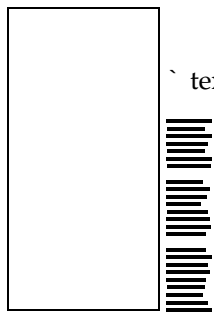
```
\output{.......
      ......
      \vfill
      \hrule 4cc
      \vskip6dd
      \box\a
      ...
      }
```

Please keep in mind that the skip will only take back the pagegoal; it does not appear anywhere, unless you use it explicitly, as we did in our output routine.

There is another value coming with inserts. A dimen giving the maximum insertion per page. This is a very interesting value that relates to our story about books of the lawyer's faculty. With this dimen we can restrict the amount of footnotes in a page to a certain amount. I.e. a `\dimen\a=.5\vsize` will restrict footnotes to half of the pageheight. If the footnotes get longer than this, it will be split and and the split-rest saved for the next page.

This needs no further explanation because it turns out to work properly without any interaction.

The more interesting parameter for inserts is the `\count`. This value gives the magnification 1000 times the factor, the pagegoal will be affected. Where can this be used? Let's take a very nice page layout rarely seen but highly recommendable.




```
output={\hbox{\vbox to\vsize{\unvbox255}
          \hskip12dd
          \box\F}}
```

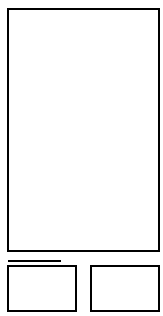
It actually is a very interesting layout and, by the way, good style. Unfortunately this kind of footnoting was lost due to "full featured" typesetting computer programs. The parameters for the insert-command now get interesting values:

```
\newinsert\F
\count\F=0
\skip\F=0pt
\dimen\F=\vsize
```

The magnification count now is 0, because our pagegoal should never be affected by the footnote-insert. Actually they are not part of the pagebody at all. The skip is 0pt for the same reason, whereas the `\dimen\F` is `\vsize`, because footnotes may easily build a complete second column as high as `\vsize`.

One may easily get misled by this example that marginal notes may be handled this way, but this is no solution for applications of that kind.

There is another interesting way to typeset footnotes.



```
\output={\vbox to\vsize{\unvbox255
                        \vfill
                        \hrule to 12cc
                        \vskip 6pt
                        \hbox to\hsize{%
                         \vsplit\a to.5\ht\a
                         \hfill
                         \box\a}}}
```

In this case the parameters are as follows:

```
\count\a=500
\dimen\a=.5\vsize
\skip\a=24dd
```

Splitting the footnote into two boxes necessarily diminishes the pagegoal by just half of the height of the collected footnote material.

You may immediately think of other applications using the same mechanism, e.g. inserts for pictures on top or on bottom of text. Here is an example to show how it works:

```
\output=\vboxto\vsize{\box\P
              \vfill
              \unvbox255}}
......Text.........
...\insert\P{\vbox to4cm{}}
```

All these examples illustrate pretty well the power of TEX's footnote algorithm. Footnotes are awkward to handle and to assemble, as a consequence the classic way to work with footnotes is fairly different under professional conditions.

1.) The footnote material is not directly inserted in the text. A normal typist will – above all for ease of use – first type the main text and then type the footnotes producing two files.

2.) footnotes come in layouts that are much more difficult than the examples above.

To illuminate this a bit, here a layout for this weekly lawyer's review magazine:

This two column layout has a few very subtle features driving even the most skilled assembler crazy. The layout demands the footnotes to appear on the left bottom on the left column. Footnotes references appearing on both columns are bound to find the referenced text complete on the same page. So no footnote breaking. When the article ends the text and footnote material have to form columns with equal length and the next article has to fit with at least three lines for each column together with related footnotes on the page.

This job is a perfect example, where a computer program is much faster and even better than any human being can be.

The task is very challenging, it does not represent a page assembly with galleys, but due to the article headlines a solution that goes beyond galley cutting.

Let's step the conditions through.

The layout for a first single page is as follows

```
\output={\if\C H
      \setbox\HL=\vbox{\unvbox255}
      \if\C L
      \setbox\Left=\vbox to\vsize{\unvbox255}
      \global\C R
      \else
      \setbox\Right=\vbox to\vsize{\unvbox255%
                          \vfill
                          \hrule4cc
                          \vskip6dd
                          \box\F}
%     assemble page
      \vbox{\hrule
          \vskip3\baselineskip
          \Headline
          \hbox to\fullhsize{\box\Left\hfill\box\Right}
          \vskip\baselineskip
          \hrule
          \vskip4dd
          \hbox to\hsize{Lawyers Gazette\hfill\the\count0}}}
```

What at the end of and article?

There are two cases again:

1.) the left column end before it is filled entirely

2.) the left column has been filled and the right column fails to get filled.

In the first case we may do the following:

```
\setbox\End=\vbox{\unvbox\255
        \vskip\baselinskip
        \hrule 4cc
        \vskip6dd
        \unvbox\F}
```

Now the footnotes and the left column are combined in the proper way in the box `\End`. The resulting box may easily be split in two equal parts.

```
\hbox to\fullhsize{%
 \split\End to .5\End
 \hfill%
 \box\End}
```

This is a pretty simple solution, which always works unless the footnotes are higher than half of the complete boxheight `\End`. But this may be unavoidable in some cases. When this comes true, there are a few ways to juggle around with the text, for instance typesetting with a different interwordspacing thus hoping for a different amount of lines avoiding the unwanted effect. There is no general solution to this problem. There are a few ways to cope with something like this. a.) fine tune the baselineskip, b.) get rid of some footnotes c.) decrease or increase the footnoteskip. But after all this effect is something that is very hard to avoid automatically.

We still have to deal with the second case for the rest of the text when the article finishes. The second case is, that the left column already was full, whereas the right column has not been completely filled.

```
\setbox\End=\vbox{\unvbox\Left%
            \unvbox\255
            \baselinskip
            \hrule 4cc
            \vskip6dd
            \unvbox\F}
```

In this case now the probability is very high, that the new article will not fit on the same page any longer. As a general rule the page now may be ended here and the new article starts at a new page. But to make sure we can check if the last column by coincidence just contains two lines and the next article would easily fit on the same page.

```
\hbox to\fullhsize{\split\End to.5\End
            \hfill
            \copy\End}
```

With the copy command the box `\End` still is available for further examination. With this we can measure the height of `\box\End` and use

the result to settle the appropriate questions, about whether or not the headline of the new article still would fit.

The final algorithm turns out to be something like this:

1.) when the article starts on a new page, collect the article headline and then collect left and right columns together with the footnotes and assemble the page.

2.) when the article ends and the left column is only partially filled, split the column together with it's footnotes and place them next to each other. Read the new article because it will fit on the same page.

3.) when the article ends with a right column, put them back together with all footnotes, split the resulting box and check the box-height. When the box height allows another article to be put on the same page, then do it, else start a new page.

In order to fulfill highest quality demands, we now could even reread articles in case they do not fit neatly into the pages. For example, if the article ends and leaves to much white space on a page and a new article may not fit into that white space, we may reread the complete article and retypeset it with a different interword spacing.

The interword spacing may for example vary from .9 times the width of an "i" in the used font and 1.3 times it's width.

We may construct the following loop to check if our article will fit neatly:

```
\spaceskip=.9\wd0 plus 1.3\wd0 minus .3\wd0
\xpaceskip=\spaceskip
%
\loop
\setbox\all=\vbox{\input article1 }
\advance\spaceskip by 1pt
\xpaceskip=\spaceskip
\ifdim\all > Newarticle\repeat
```

In this way the complete magazine may be typeset automatically without any interaction by stating the pertinent rules.

This example proves, that the programming of TEX is not different to any other programming task. The most important prerequisite focus on the rules how things should be done.

TEX is not a preprogrammed expert system, and this is its strength, because almost any application may be programmed with TEX.

As we already mentioned, the typist will not normally type the bodytext and footnote text in one file. Usually you get two files. To solve the problem properly we have to write some kind of TEX program, that is able to read records from a file.

Finally I'd like to turn my attention to one of the most important and fastest growing application fields for TEX.

Typesetting material from databases

Material from databases comprises almost anything. Computers have become ubiquitous. From the early days of hammer-printing and high speed chain printers we can produce nice looking output on laser printer today. Parallely there has always been the area of professional typesetting. Special computers have been invented and built in order to get typesetting done easier, faster, and more convenient. Things tend to come together again. The normal computer applications, as there are computer aided design, mathematics, physics all produce output that is supposed to be read by other people. Almost any computer output on paper represents the final result of the work. No wonder if typesetting and other fields of computing get closer and closer.

But not only the publication of scientific material is the matter in question, but also all these incredible heaps of data that mankind collect day and night in huge databases. Dealing with flight schedules; train departure hours, telephone directories, or all 0 rhesus negative people in the world ready to donate their kidney. All this sometimes very often needs to be presented and packed in a neat typeset form.

The power of TEX is right here, more than in any other field. TEX can read data in ASCII format and it can separate data in fields and records and therefore TEX is the best-balanced racket for this kind of game.

The only important knowledge we need to typeset material from a database is how to handle data in a format that is record oriented.

Lets just take a single entry from a tax table:

```
12345,45
IV*21239*II-93200*89400*44324*52332*23323*28323*29923*29323;
IV*12380*II-93000*87396*43724*23832*55233*28323*23923*28823;
IV*21239*II-93200*89400*44324*52332*23323*28323*29923*29323;
IV*12380*II-93000*87396*43724*23832*55233*28323*23923*28823;
```

The record is dominated by a bunch of numbers representing many different parameters for a macro, that is supposed to typeset this kind of material. TEX does not offer more than 10 parameters for one macro. Therefore this chapter deals above all with all possible ways how to increase the number of parameters for a macro.

In the above example, we might consider the following macro to handle this kind of material with TEX.

```
\entry 12\s 345,45;
IV*21239*II-93200*89400*44324*52332*23323*28323*29923*29323;
IV*12380*II-93000*87396*43724*23832*55233*28323*23923*28823;
IV*21239*II-93200*89400*44324*52332*23323*28323*29923*29323;
IV*12380*II-93000*87396*43724*23832*55233*28323*23923*28823;
```

The definition of entry may now be:

```
\def\entry#1;#2;#3;#4;#5;
```

The huge bundle of different numbers now gets divided in five packages:

```
#1->12\s 345,45
#2->IV*21239*II-93200*89400*44324*52332*23323*28323*29923*29323
#3->IV*12380*II-93000*87396*43724*23832*55233*28323*23923*28823
#4->IV*21239*II-93200*89400*44324*52332*23323*28323*29923*29323
#5->IV*12380*II-93000*87396*43724*23832*55233*28323*23923*28823
```

A second macro may now be defined, that is able to separate the numbers:

```
\def\elmnt#1*#2*#3*#4*#5*#6*#7*#8*#9*{....
```

which apperantly wouldn't work, because there are more than 9 elements in a single line of numbers. The only way to cope with things like this is to separate parameters stepwise.

```
\def\el#1*#2;
```

This would require the following calling sequence:

```
\def\entry#1;#2;#3;#4;#5;{\el#2;
                  .
                  .}
```

What happens apparently is the following. The parameter #2 from entry reaches from the semicolon ; to the next colon resulting in the following string:

```
IV*21239*II-93200*89400*44324*52332*23323*28323*29923*29323
```

If we call \el with this string it will be torn apart, because TEX reads the string up to a asterisk (*) to form the first parameter, giving IV and read from the asterisk to the colon (;) to form the second parameter. The first macro tt\entry already has stripped the colon, therefore the colon has to be set again, this is why \el will be used as \el#2; In this way we may strip down element by element up to the point, where less than 9 elements are left and you may introduce some macro like

```
\def\final#1*#2*#3*#4*#5*#6*#7*#8*#9
```

This may often be a very good solution, especially in our case, if the structural appearance of the data material is reflected in this kind of method. Stripping not only leads to a separation of the single elements, but also results in different typesetting actions that are directly put into the according macros.

But we might as well state a general macro that may strip any number of fields from a record. This macro should have the following syntax:

```
\strip14
 IV*21239*II-93200*89400*44324*52332*23323*28323*29923*29323;
```

is supposed to mean: strip the 14th field out of the record.

```
\def\strip#1 #2*#3;{\count0=#1\advance\count0 by -1
             \ifnum\count0
              \strip\count0 #3;
             \fi
             \Field={#2}}
```

The definition of split is a recursive algorithm, that successively strips element for element from a string that consists of characters separated by asterisks.