CHAPTER 4
NROFF AND TROFF

The vi editor lets you edit text, but it is not much good at formatting. A text file such as program source code might be formatted with a simple program like pr, which inserts a header at the top of every page and handles pagination, but otherwise prints the document exactly as it appears in the file. But for any application requiring the preparation of neatly formatted text, you will use the nroff ("en-roff") or troff ("tee-roff") formatting program.

These programs are used to process an input text file, usually coded or "marked up" with formatting instructions. When you use a *wysiwyg* program like most word processors, you use commands to lay out the text on the screen as it will be laid out on the page. With a markup language like that used by nroff and troff, you enter commands into the text that tell the formatting program what to do.

Our purpose in this chapter is twofold. We want to introduce the basic formatting codes that you will find useful. But at the same time, we want to present them in the context of what the formatter is doing and how it works. If you find this chapter rough-going—especially if this is your first exposure to nroff/ troff— skip ahead to either Chapter 5 or Chapter 6 and become familiar with one of the macro packages, ms or mm; then come back and resume this chapter. We assume that you are reading this book because you would like more than the basics, that you intend to master the complexities of nroff/ troff. As a result, this chapter is somewhat longer and more complex than it would be if the book were an introductory user's guide.

Conventions

To distinguish input text and requests shown in examples from formatter output, we have adopted the convention of showing "page corners" around output from nroff or troff. Output from nroff is shown in the same constant-width typeface as other examples:

```
Here is an example of nroff output.
```

Output from troff is shown in the same typeface as the text, but with the size of the type reduced by one point, unless the example calls for an explicit type size:

Here is an example of troff output.

In representing output, compromises sometimes had to be made. For example, when showing nroff output, we have processed the example separately with nroff, and read the results back into the source file. However, from there, they have been typeset in a constant-width font by troff. As a result, there might be slight differences from true nroff output, particularly in line length or page size. However, the context should always make clear just what is being demonstrated.

□ What the Formatter Does □

Take a moment to think about the things you do when you format a page on a *wysiwyg* device such as a typewriter:

- You set aside part of the page as the text area. This requires setting top, bottom, left, and right margins.
- You adjust the lines that you type so they are all approximately the same length and fit into the designated text area.
- You break the text into syntactic units such as paragraphs.
- You switch to a new page when you reach the bottom of the text area.

Left to themselves, `nroff` or `troff` will do only one of these tasks: they will adjust the length of the lines in the input file so that they come out even in the output file. To do so, they make two assumptions:

- They assume that the line length is 6.5 inches.
- They assume that a blank line in the input signals the start of a new paragraph. The last line of the preceding text is not adjusted, and a blank line is placed in the output.

The process of filling and adjusting is intuitively obvious—we've all done much the same thing manually when using a typewriter or had it done for us by a *wysiwyg* word processor. However, especially when it comes to a typesetting program like `troff`, there are ramifications to the process of line adjustment that are not obvious. Having a clear idea of what is going on will be very useful later. For this reason, we'll examine the process in detail.

Line Adjustment

There are three parts to line adjustment: *filling*, *justification*, and *hyphenation*. Filling is the process of making all lines of text approximately equal in length. When working on a typewriter, you do this automatically, simply by typing a carriage return when the line is full. Most word-processing programs automatically insert a carriage return at the end of a line, and we have seen how to set up `vi` to do so as well.

However, `nroff` and `troff` ignore carriage returns in the input except in a special "no fill" mode. They reformat the input text, collecting all input lines into even-length output lines, stopping only when they reach a blank `1`ine or (as we shall see shortly) a formatting instruction that tells them to stop. Lines that begin with one or more blank spaces are not filled, but trailing blank spaces are trimmed. Extra blank spaces between words on the input line are preserved, and the formatter adds an extra blank space after each period, question mark, or exclamation point.

Justification is a closely related feature that should not be confused with filling. Filling simply tries to keep lines approximately the same length; justification adjusts the space between words so that the ends of the lines match exactly.

By default, `nroff` and `troff` both fill and justify text. Justification implies filling, but it is possible to have filling without justification. Let's look at some examples. First, we'll look at a paragraph entered in `vi`. Here's a paragraph from the letter you entered in the last chapter, modified so that it offers to prepare not just a user's guide for the Alcuin illuminated lettering software, but a reference manual as well. In the course of making the changes, we've left a short line in the middle of the paragraph.

```
In our conversation last Thursday, we discussed a
documentation project that would produce a user's guide
and reference manual
for the Alcuin product. Yesterday, I received the product
demo and other materials that you sent me.
```

Now, let's look at the paragraph after processing by `nroff`:

```
In our  conversation  last Thursday, we discussed  a
documentation project  that  would  produce a user's
guide and reference manual  for the Alcuin  product.
Yesterday, I  received  the  product  demo and other
materials that you sent me.
```

The paragraph has been both filled and justified. If the formatter were told to fill, but not to justify, the paragraph would look like this:

```
In our conversation last Thursday, we discussed a
documentation project that would produce a user's guide
and reference manual for the Alcuin product. Yesterday,
I received the product demo and other materials that
you sent me.
```

As you can see, `nroff` justified the text in the first example by adding extra space between words.

Most typewritten material is filled but not justified. In printer's terms, it is typed *ragged right*. Books, magazines, and other typeset materials, by contrast, are usually *right justified*. Occasionally, you will see printed material (such as ad copy) in which the right end of each line is justified, but the left end is ragged. It is for this reason that we usually say that text is *right* or *left justified*, rather than simply *justified*.

When it is difficult to perform filling or justification or both because a long word falls at the end of a line, the formatter has another trick to fall back on (one we are all familiar with)—hyphenation.

The `nroff` and `troff` programs perform filling, justification, and hyphenation in much the same way as a human typesetter used to set cold lead type. Human typesetters used to assemble a line of type by placing individual letters in a tray until each line was filled. There were several options for filling as the typesetter reached the end of the line:

- The next word might fit exactly.
- The next word might fit if the typesetter squeezed the words a little closer together.
- The next word could be hyphenated, with part put on the current 1ine and part on the next line.

If, in addition to being filled, the text was to be justified, there was one additional issue: after the line was approximately the right length, space needed to be added between each word so that the line length came out even.

Just like the human typesetter they replace, `nroff` and `troff` assemble one line of text at a time, measuring the length of the line and making adjustments to the spacing to make the line come out even (assuming that the line is to be justified). Input lines are collected into a temporary storage area, or *buffer*, until enough text has been collected for a single output line. Then that line is output, and the next line collected.

It is in the process of justification that you see the first significant difference between the two programs. The `nroff` program was designed for use with typewriter-like printers; `troff` was designed for use with phototypesetters.

A typewriter-style printer has characters all of the same size—an *i* takes up the same amount of space as an *m*. (Typical widths are 1/10 or 1/12 inch per character). And although some printers (such as daisywheel printers) allow you to change the style of type by changing the daisywheel or thimble, you can usually have only one typeface at a time.

A typesetter, by contrast, uses typefaces in which each letter takes up an amount of space proportional to its outline. The space allotted for an *i* is quite definitely narrower than the space allotted for an *m*. The use of variable-width characters makes the job of filling and justification much more difficult for `troff` than for `nroff`. Where `nroff` only needs to count characters, `troff` has to add up the width of each character as it assembles the line. (Character widths are defined by a "box" around the character, rather than by its natural, somewhat irregular shape).

The `troff` program also justifies by adding space between words, but because the variable-width fonts it uses are much more compact, it fits more on a line and generally does a much better job of justification.*

There's another difference as well. Left to itself, `nroff` will insert only full spaces between words—that is, it might put two spaces between one pair of words, and three between another, to fill the line. If you call `nroff` with the `-e` option, it will attempt to make all interword spaces the same size (using fractional spaces if possible). But even then, `nroff` will only succeed if the output device allows fractional spacing. The `troff` program always uses even interword spacing.

Here's the same paragraph filled and justified by `troff`:

> In our conversation last Thursday, we discussed a documentation project that would produce a user's guide and reference manual for the Alcuin product. Yesterday, I received the product demo and other materials that you sent me.

To make matters still more difficult, typeset characters come in a variety of different designs, or *fonts*. A font is a set of alphabetic, numeric, and punctuation characters that share certain design elements. Typically, fonts come in families of several related typefaces. For example, this book is typeset for the most part in the Times Roman family of typefaces. There are three separate fonts:

roman
bold
italic

Typesetting allows for the use of multiple fonts on the same page, as you can see from the mixture of fonts throughout this book. Sometimes the fonts are from the same family, as with the Times Roman, Times Bold, and Times Italic just shown. However, you can see other fonts, such as Helvetica, in the running headers on each page. Bold and italic fonts are generally used for emphasis; in computer books such as this, a constant-width typewriter font is used for examples and other "computer voice" statements.

Even within the same font family, the width of the same character varies from font to font. For example, a bold "**m**" is slightly wider than a Roman "m."

To make things still more complicated, the same font comes in different sizes. If you look at this book, you will notice that the section headings within each chapter are slightly larger for emphasis. Type sizes are measured in units called *points*. We'll talk more about this later, but to get a rough idea of what type sizes mean, simply look at the current page. The body type of the book is 10-point Times Roman; the next heading is 12-point Times Bold. The spacing between lines is generally proportional to the point size, instead of fixed, as it is with `nroff`.

The `troff` program gets information about the widths of the various characters in each font from tables stored on the system in the directory `/usr/lib/font`. These tables tell `troff` how far to move over after it has output each character on the line.

We'll talk more about `troff` later. For the moment, you should be aware that the job of the formatting program is much more complicated when typesetting than it is when preparing text for typewriter-style

*The very best typesetting programs have the capability to adjust the space between individual characters as well. This process is called *kerning*. SoftQuad Publishing Software in Toronto sells an enhanced version of `troff` called `SQroff` that does support kerning.

printers.

▫ Using **nroff** ▫

As mentioned previously, left to themselves, `nroff` and `troff` perform only rudimentary formatting. They will fill and justify the text, using a default line length of 6.5 inches, but they leave no margins, other than the implicit right margin caused by the line length. To make this clearer, let's look at the sample letter from the last chapter (including the edit we made in this chapter) as it appears after formatting with `nroff`.

First, let's look at how to invoke the formatter. The `nroff` program takes as an argument the name of a file to be formatted:

```
$ nroff letter
```

Alternatively, it can take standard input, allowing you to preprocess the text with some other program before formatting it:

```
$ tbl report | nroff
```

There are numerous options to `nroff`. They are described at various points in this book (as appropriate to the topic) and summarized in Appendix B.

One basic option is `−T`, which specifies the terminal (printer) type for which output should be prepared. Although `nroff` output is fairly straightforward, some differences between printers can significantly affect the output. (For example, one printer may perform underlining by backspacing and printing an underscore under each underlined letter, and another may do it by suppressing a newline and printing the underscores in a second pass over the line). The default device is the Teletype Model 37 terminal—a fairly obsolete device. Other devices are listed in Appendix B. If you don't recognize any of the printers or terminals, the safest type is probably `lp`:

```
$ nroff −Tlp file
```

In examples in this book, we will leave off the `−T` option, but you may want to experiment, and use whichever type gives the best results with your equipment.

Like most UNIX programs, `nroff` prints its results on standard output. So, assuming that the text is stored in a file called `letter`, all you need to do is type:

```
$ nroff letter
```

A few moments later, you should see the results on the screen. Because the letter will scroll by quickly, you should pipe the output of `nroff` to a paging program such as `pg` or `more`:

```
$ nroff letter | pg
```

or out to a printer using `lp` or `lpr`:

```
$ nroff letter | lp
```

▫ Using **troff** ▫

The chief advantage of `troff` over `nroff` is that it allows different types of character sets, or fonts, and so lets you take full advantage of the higher-quality printing available with typesetters and laser printers. There are a number of requests, useful only in `troff`, for specifying fonts, type sizes, and the vertical spacing between lines. Before we describe the actual requests though, we need to look at a bit of history.

The `troff` program was originally designed for a specific typesetter, the Wang C/A/T. Later, it was modified to work with a wide range of output devices. We'll discuss the original version of `troff` (which is still in use at many sites) first, before discussing the newer versions. The C/A/T typesetter was designed in such a way that it could use only four fonts at one time.

(Early phototypesetters worked by projecting light through a film containing the outline of the various characters. The film was often mounted on a wheel that rotated to position the desired character in front of the light source as it flashed, thus photographing the character onto photographic paper or negative film. Lenses enlarged and reduced the characters to produce various type sizes. The C/A/T typesetter had a wheel divided into four quadrants, onto which one could mount four different typefaces).

Typically, the four fonts were the standard (roman), bold, and italic fonts of the same family, plus a "special" font that contained additional punctuation characters, Greek characters (for equations), bullets, rules, and other nonstandard characters. Figure 4-1 shows the characters available in these standard fonts.

The Coming of `ditroff`

Later, `troff` was modified to support other typesetters and, more importantly (at least from the perspective of many readers of this book), laser printers. The later version of `troff` is often called `ditroff` (for device-independent `troff`), but many UNIX systems have changed the name of the original `troff` to `otroff` and simply call `ditroff` by the original name, `troff`.

The `ditroff` program has not been universally available because, when it was developed, it was "unbundled" from the basic UNIX distribution and made part of a separate product called *Documenter's Workbench* or *DWB*. UNIX system manufacturers have the option not to include this package, although increasingly, they have been doing so. Versions of DWB are also available separately from third party vendors.

The newer version of `troff` allows you to specify any number of different fonts. (You can mount fonts at up to ten imaginary "positions" with `.fp` and can request additional fonts by name).
There may also be different font sizes available, and there are some additional commands for line drawing (`ditroff` can draw curves as well as straight lines). For the most part, though, `ditroff` is very similar to the original program, except in the greater flexibility it offers to use different output devices.

One way to find out which version of `troff` you have on your system (unless you have a program explicitly called `ditroff`) is to list the contents of the directory `/usr/lib/font`:

```
$ls -F /usr/lib/font
devlj/
devps/
ftB
ftI
ftR
ftS
```

If there are one or more subdirectories whose name begins with the letters `dev`, your system is using `ditroff`. Our system supports both `ditroff` and `otroff`, so we have both a device subdirectory (for `ditroff`) and font files (for `otroff`) directly in `/usr/lib/font`.

We'll talk more about font files later. For the moment, all you need to know is that they contain information about the widths of the characters in various fonts for a specific output device.

Contrary to what a novice might expect, font files do not contain outlines of the characters themselves. For a proper typesetter, character outlines reside in the typesetter itself. All `troff` sends out to the typesetter are character codes and size and position information.

However, `troff` has increasingly come to be used with laser printers, many of which use *downloadable fonts*. An electronic image of each character is loaded from the computer into the printer's memory, typically at the start of each printing job. There may be additional "font files" containing character outlines in this case, but these files are used by the software that controls the printer, and have nothing to do with `troff` itself. In other cases, font images are stored in ROM (read-only memory) in the printer.

If you are using a laser printer, it is important to remember that `troff` itself has nothing to do with the actual drawing of characters or images on the printed page. In a case like this, `troff` simply formats the page, using tables describing the widths of the characters used by the printer, and generates instructions about page layout, spacing, and so on. The actual job of driving the printer is handled by another program, Times Roman

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + - . , / : ; = ? [ ] |
• □ — – _ ¼ ½ ¾ fi fl ° † ´ ¢ ® ©

Times Italic

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
*! $ % & ( ) ‘ ’ * + - . , / : ; = ? [ ] |*
• □ — – _ ¼ ½ ¾ fi fl ° † ´ ¢ ® ©

Times Bold

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + - . , / : ; = ? [ ] |
• □ — – _ ¼ ½ ¾ fi fl ° † ´ ¢ ® ©

Special Mathematical Font

" ´ \ ^ _ ` ~ / ⟨ ⟩ { } # @ + − = ∗
α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ σ ς τ υ φ χ ψ ω
Γ Δ Θ Λ Ξ Π Σ ϒ Φ Ψ Ω
√ ‾ ≥ ≤ ≡ ~ ≠ → ← ↑ ↓ × ÷ ± ∪ ∩ ⊂ ⊃ ⊆ ⊇ ∞ ∂
§ ∇ ¬ ∫ ∝ ∅ ∈ ‡ ☞ ☜ | ○ ⌈ ⌊ ⌉ ⌋ { } | ⌊ ⌋⌈ ⌉|

Figure 4.1 The Four Standard Fonts

generally referred to as a *printer driver* or `troff` *postprocessor*.

To use `troff` with such a postprocessor, you will generally need to pipe the output of `troff` to the postprocessor and from there to the print spooler:

$ **troff** *file* **|** *postprocessor* **| lp**

If you are using the old version of `troff`, which expects to send its output directly to the C/A/T typesetter, you need to specify the `-t` option, which tells `troff` to use standard output. If you don't, you will get the message:

```
Typesetter busy.
```

(Of course, if by any chance you *are* connected to a C/A/T typesetter, you don't need this option. There are several other options listed in Appendix B that you may find useful). When you use `ditroff`, on the other hand, you will need to specify the `-T` command-line option that tells it what device you are using. The postprocessor will then translate the device-independent `troff` output into instructions for that particular type of laser printer or typesetter. For example, at our site, we use `troff` with an Apple LaserWriter and Pipeline Associates' `devps` postprocessor, which translates `troff` output for the LaserWriter. Our command line looks something like this:

$ **ditroff -Tps** *files* **| devps | lp**

You can print the same file on different devices, simply by changing the `-T` option and the postprocessor. For example, you can print drafts on a laser printer, then switch to a typesetter for final output without making extensive changes to your files. (To actually direct output to different printers, you will also have to specify a printer name as an option to the `lp` command. In our generic example, we simply use `lp` without any options, assuming that the appropriate printer is connected as the *default* printer).

Like all things in life, this is not always as easy as it sounds. Because the fonts used by different output devices have different widths even when the nominal font names and sizes are the same, pagination and line breaks may be different when you switch from one device to another.

The job of interfacing `ditroff` to a wide variety of output devices is becoming easier because of the recent development of industry-wide *page description languages* like Adobe Systems' PostScript, Xerox's Interpress, and Imagen's DDL. These page description languages reside in the printer, not the host computer, and provide a device-independent way of describing placement of characters and graphics on the page.

Rather than using a separate postprocessor for each output device, you can now simply use a postprocessor to convert `troff` output to the desired page description language. For example, you can use Adobe Systems' TranScript postprocessor (or an equivalent postprocessor like `devps` from Pipeline Associates) to convert `troff` output to PostScript, and can then send the PostScript output to any one of a number of typesetters or laser printers.

From this point, whenever we say `troff`, we are generally referring to `ditroff`. In addition, although we will continue to discuss `nroff` as it differs from `troff`, our emphasis is on the more capable program. It is our opinion that the growing availability of laser printers will make `troff` the program of choice for almost all users in the not too distant future.

However, you can submit a document coded for `troff` to `nroff` with entirely reasonable results. For the most part, formatting requests that cannot be handled by `nroff` are simply ignored. And you can submit documents coded for `nroff` to `troff`, though you will then be failing to use many of the characteristics that make `troff` desirable.

▫ The Markup Language ▫

The `nroff` and `troff` markup commands (often called *requests*) typically consist of one or two lowercase letters and stand on their own line, following a period or apostrophe in column one. Most requests are reasonably mnemonic. For example, the request to leave space is:

```
.sp
```

There are also requests that can be embedded anywhere in the text. These requests are commonly called *escape sequences*. Escape sequences usually begin with a backslash (`\`). For example, the escape sequence `\l` will draw a horizontal line. Especially in `troff`, escape sequences are used for line drawing or for printing various special characters that do not appear in the standard ASCII character set. For instance, you enter `\(bu` to get •, a bullet.

There are three classes of formatting instructions:

- Instructions that have an immediate one-time effect, such as a request to space down an inch before outputting the next line of text.
- Instructions that have a persistent effect, such as requests to set the line length or to enable or disable justification.
- Instructions that are useful for writing *macros*. There is a "programming language" built into the formatter that allows you to build up complex requests from sequences of simpler ones. As part of this language there are requests for storing values into variables called *strings* and *number registers*, for testing conditions and acting on the result, and so on.

For the most part, we will discuss the requests used to define macros, strings, and number registers later in this book.

At this point, we want to focus on understanding the basic requests that control the basic actions of the formatter. We will also learn many of the most useful requests with immediate, one-time effects. Table

4-1 summarizes the requests that you will use most often.

Table 4.1 Basic* `nroff/troff` *Requests

Request	Meaning	Request	Meaning
`.ad`	Enable line adjustment	`.na`	No justification of lines
`.br`	Line break	`.ne`	Need lines to end of page
`.bp`	Page break	`.nf`	No filling of lines
`.ce`	Center next line	`.nr`	Define and set number register
`.de`	Define macro	`.po`	Set page offset
`.ds`	Define string	`.ps`	Set point size
`.fi`	Fill output lines	`.so`	Switch to source file and return
`.ft`	Set current font	`.sp`	Space
`.in`	Set indent	`.ta`	Set tab stop positions
`.ls`	Set double or triple spacing	`.ti`	Set temporary indent
`.ll`	Specify line length	`.vs`	Set vertical line spacing

Looking at `nroff` Output

When we discussed the basic operations of the text formatter, we saw that `nroff` and `troff` perform rudimentary formatting. They will fill and justify the text, using a default line length of 6.5 inches, but they leave no margins, other than the implicit right margin caused by the line length.

To make this clearer, let's look at the sample letter from the last chapter as it appears after formatting with `nroff`, without any embedded requests, and without using any macro package. From Figure 4-2, you can see immediately that the formatter has adjusted all of the lines, so that they are all the same length—even in the address block of the letter, where we would have preferred them to be left as they were. Blank lines in the input produce blank lines in the output, and the partial lines at the ends of paragraphs are not adjusted.

The most noticeable aspect of the raw formatting is a little difficult to reproduce here, though we've tried. No top or left margin is automatically allocated by `nroff`.

▫ Turning Filling On and Off ▫

Even though filling of uneven text lines resulting from editing is probably the most basic action we want from the formatter, it is not always desirable. For example, in our letter, we don't want the address block to be filled. There are two requests we could use to correct the problem: `.br` (*break*) and `.nf` (*no fill*).

A `.br` request following a line outputs the current contents of the line buffer and starts the next line, even though the buffer is not yet full. To produce a properly formatted address block, we could enter the following requests in the file:

```
Mr. John Fust
.br
Vice President, Research and Development
.br
Gutenberg Galaxy Software
.br
Waltham, Massachusetts 02159
```

Each individual input line will be output without filling or justification. We could also use the `.nf` request, which tells `nroff` to stop filling altogether. Text following this request will be printed by the formatter exactly as it appears in the input file. Use this request when you want text to be laid out as it was typed in.

Because we do want the body of the letter to be filled, we must turn filling back on with the `.fi` (*fill*) request:

```
                                            April 1, 1987
.nf
Mr. John Fust
Vice President, Research and Development
Gutenberg Galaxy Software
Waltham, Massachusetts 02159
.fi
Dear Mr. Fust:
```

```
                              April 1, 1987

Mr.  John  Fust  Vice  President,   Research   and
Development  Gutenberg  Galaxy  Software  Waltham,
Massachusetts 02159

Dear Mr. Fust:

In our conversation last Thursday, we discussed  a
documentation  project that would produce a user's
guide and reference manual for the Alcuin product.
Yesterday,  I  received the product demo and other
materials that you sent me.  After studying  them,
I want to clarify a couple of points:

Going through a demo session gave me a much better
understanding  of the product.  I confess to being
amazed  by  Alcuin.   Some  people  around   here,
looking  over  my shoulder, were also astounded by
the illustrated manuscript I produced with Alcuin.
One  person,  a student of calligraphy, was really
impressed.

Tomorrow, I'll start putting  together  a  written
plan   that   presents   different  strategies  for
documenting the Alcuin  product.  After  I  submit
this  plan,  and  you  have had time to review it,
let's arrange a meeting at your company to discuss
these strategies.

Thanks again for giving us the opportunity to  bid
on  this  documentation  project.   I  hope we can
decide upon a strategy and get started as soon  as
possible in order to have the manual ready in time
for first customer ship. I look forward to meeting
with you towards the end of next week.

                                   Sincerely,



                                   Fred Caslon
```

Figure 4.2 A Raw `nroff`-formatted File

If you look carefully at the previous example, you will probably notice that we entered the two formatting requests on blank lines in the letter. If we were to format the letter now, here is what we'd get:

```
                                             April 1, 1987
Mr. John Fust
Vice President, Research and Development
Gutenberg Galaxy Software
Waltham, Massachusetts 02159
Dear Mr. Fust:
```

As you may notice, we've lost the blank lines that used to separate the date from the address block, and the address block from the salutation. Lines containing formatting requests do not result in any space being output (unless they are spacing requests), so you should be sure not to inadvertently replace blank lines when entering formatting codes.

▫ Controlling Justification ▫

Justification can be controlled separately from filling by the `.ad` (*adjust*) request. (However, filling must be on for justification to work at all). You can adjust text at either margin or at both margins.

Unlike the `.br` and `.nf` requests introduced, `.ad` takes an *argument*, which specifies the type of justification you want:

`l`	adjust left margin only
`r`	adjust right margin only
`b`	adjust both margins
`c`	center filled line between margins

There is another related request, `.na` (*no adjust*). Because the text entered in a file is usually left justified to begin with, turning justification off entirely with `.na` produces similar results to `.ad l` in most cases.

However, there is an important difference. Normally, if no argument is given to the `.ad` request, both margins will be adjusted. That is, `.ad` is the same as `.ad b`. However, following an `.na` request, `.ad` reverts to the value last specified. That is, the sequence:

```
.ad r
Some text
.ad l
Some text
.ad
Some text
```

will adjust both margins in the third block of text. However, the sequence:

```
.ad r
Some text
.na
Some text
.ad
Some text
```

will adjust only the right margin in the third block of text.

It's easy to see where you would use `.ad b` or `.ad l`. Let's suppose that you would like a ragged margin for the body of your letter, to make it look more like it was prepared on a typewriter. Simply follow the `.fi` request we entered previously with `.ad l`.

Right-only justification may seem a little harder to find a use for. Occasionally, you've probably seen ragged-left copy in advertising, but that's about it. However, if you think for a moment, you'll realize that it is also a good way to get a single line over to the right margin.

For example, in our sample letter, instead of typing all those leading spaces before the date (and having it fail to come out flush with the margin anyway), we could enter the lines:

```
.ad r
April 1, 1987
.ad b
```

As it turns out, this construct won't *quite* work. If you remember, when filling is enabled, `nroff` and `troff` collect input in a one-line buffer and only output the saved text when the line has been filled. There are some non-obvious consequences of this that will ripple all through your use of `nroff` and `troff`. If you issue a request that temporarily sets a formatting condition, then reset it before the line is output, your original setting may have no effect. *The result will be controlled by the request that is in effect at the time the line is output, not at the time that it is first collected in the line buffer.*

Certain requests cause implicit line breaks (the equivalent of carriage returns on a typewriter) in the output, but others do not. The `.ad` request does not cause a break. Therefore, a construction like:

```
.ad r
April 1, 1987
.ad b
Mr. John Fust
```

will result in the following output:

```
April 1, 1987 Mr. John Fust
```

and not:

```
                                        April 1, 1987
Mr. John Fust
```

To make sure that you get the desired result from a temporary setting like this, be sure to follow the line to be affected with a condition that will cause a break.* For instance, in the previous example, you would probably follow the date with a blank line or an `.sp` request, either of which will normally cause a break. If you don't, you should put in an explicit break, as follows:

```
.ad r
April 1, 1987
.br
.ad b
Mr. John Fust
```

A final point about justification: the formatter adjusts a line by widening the blank space between words. If

*The following requests cause a break:

.bp .br .ce .fi .nf .sp .in .ti

All other requests can be interspersed with text without causing a break. In addition, as discussed later, even these requests can be introduced with a special "no break" control character (`'` instead of `.`) so that they too will not cause a break.

you do not want the space between two words adjusted or split across output lines, precede the space with a backslash. This is called an *unpaddable space*.

There are many obscure applications for unpaddable spaces; we will mention them as appropriate. Here's a simple one that may come in handy: `nroff` and `troff` normally add two blank spaces after a period, question mark, or exclamation point. The formatter can't distinguish between the end of a sentence and an abbreviation, so if you find the extra spacing unaesthetic, you might follow an abbreviation like Mr. with an unpaddable space: `Mr.\ John Fust`.

▫ Hyphenation ▫

As pointed out previously, hyphenation is closely related to filling and justification, in that it gives `nroff` and `troff` some additional power to produce filled and justified lines without large gaps.

The `nroff` and `troff` programs perform hyphenation according to a general set of rules. Occasionally, you need to control the hyphenation of particular words. You can specify either that a word not be hyphenated or that it be hyphenated in a certain way. You can also turn hyphenation off entirely.

Specifying Hyphenation for Individual Words

There are two ways to specify that a word be hyphenated a specific way: with the `.hw` request and with the special hyphenation indicator `\%`.

The `.hw` (*hyphenate word*) request allows you to specify a small list of words that should be hyphenated a specific way. The space available for the word list is small (about 128 characters), so you should use this request only for words you use frequently, and that `nroff` and `troff` hyphenate badly.

To use `.hw`, simply specify the word or words that constitute the exception list, typing a hyphen at the point or points in the word where you would like it to be hyphenated.

```
.hw hy-phen-a-tion
```

You can specify multiple words with one `.hw` request, or you can issue multiple `.hw` requests as you need them.

However, if it is just a matter of making sure that a particular instance of a word is hyphenated the way you want, you can use the hyphenation indication character sequence `\%`. As you type the word in your text, simply type the two characters `\%` at each acceptable hyphenation point, or at the front of the word if you don't want the word to be hyphenated at all:

`\%acknowledge` the word *acknowledge* will not be hyphenated
`ac\%know\%ledge` the word *acknowledge* can be hyphenated only at the specified points

This character sequence is the first instance we have seen of a formatting request that does not consist of a request name following a period in column one. We will see many more of these later. This sequence is embedded right in the text but does not print out.

In general, `nroff` and `troff` do a reasonable job with hyphenation. You will need to set specific hyphenation points only in rare instances. In general, you shouldn't even worry about hyphenation points, unless you notice a bad break. Then use either `.hw` or `\%` to correct it.

The UNIX `hyphen` command can be used to print out all of the hyphenation points in a file formatted with `nroff` or `troff -a`.

$ **nroff** *options files* **|** **hyphen**

or:

$ **troff** *options* **-a** *files* **|** **hyphen**

If your system doesn't have the `hyphen` command, you can use `grep` instead:

```
$ nroff options files | grep '-$'
```

(The single quotation marks are important because they keep `grep` from interpreting the − as the beginning of an option).

Turning Hyphenation Off and On

If you don't want any hyphenation, use the `.nh` (*no hyphenation*) request. Even if you do this, though, you should be aware that words already containing embedded hyphens, em dashes (—), or hyphen indication characters (`\%`) will still be subject to hyphenation.

After you've turned hyphenation off, you can turn it back on with the `.hy` (*hyphenate*) request. This request has a few twists. Not only does it allow you to turn hyphenation on, it also allows you to adjust the hyphenation rules that `nroff` and `troff` use. It takes the following numeric arguments:

0	turn hyphenation off
1	turn hyphenation on
2	do not hyphenate the last line on a page
4	do not hyphenate after the first two characters of a word
8	do not hyphenate before the last two characters of a word

Specifying `.hy` with no argument is the same as specifying `.hy 1`. The other numeric values are additive. For example, `.hy 12` (`.hy 4` plus `.hy 8`) will keep `nroff` and `troff` from breaking short syllables at the beginning or end of words, and `.hy 14` will put all three hyphenation restrictions into effect.

□ Page Layout □

Apart from the adjusted address block, the biggest formatting drawback that you probably noticed when we formatted the sample letter is that there was no left or top margin. Furthermore, though it is not apparent from our one-page example, there is no bottom margin either. If there were enough text in the input file to run onto a second page, you would see that the text ran continuously across the page boundary.

In normal use, these layout problems would be handled automatically by either the `ms` or `mm` macro packages (described later). Here, though, we want to understand how the formatter itself works.

Let's continue our investigation of the `nroff` and `troff` markup language with some basic page layout commands. These commands allow you to affect the placement of text on the page. Some of them (those whose descriptions begin with the word *set*) specify conditions that will remain in effect until they are explicitly changed by another instance of the same request. Others have a one-time effect.

As shown in Table 4-2, there are two groups of page layout commands, those that affect horizontal placement of text on the page and those that affect vertical placement. A moment's glance at these requests will tell you that, before anything else, we need to talk about units.

Table 4.2 Layout Commands

Horizontal Layout	`.ll` *n*	Set the line length to *n*
	`.po` *n*	Set the left margin (page offset) to *n*
	`.in` *n*	Indent the left margin to *n*
	`.ti` *n*	Temporarily indent the left margin to *n*
	`.ce` *n*	Center the following *n* lines
Vertical Layout	`.pl` *n*	Set the page length to *n*
	`.sp` *n*	Insert *n* spaces
	`.bp` *n*	Start a new page
	`.wh` *n*	Specify *wh*en (at what vertical position on the page) to execute a command

Units of Measure

By default, most `nroff` and `troff` commands that measure vertical distance (such as `.sp`) do so in terms of a number of "lines" (also referred to as vertical spaces, or `vs`). The `nroff` program has constant, device-dependent line spacing; `troff` has variable line spacing, which is generally proportional to the point size. However, both programs do allow you to use a variety of other units as well. You can specify spacing in terms of inches and centimeters, as well as the standard printer's measures *picas* and *points*. (A pica is 1/6 of an inch; a point is about 1/72 of an inch. These units were originally developed to measure the size of type, and the relationship between these two units is not as arbitrary as it might seem. A standard 12-point type is 1 pica high).

Horizontal measures, such as the depth of an indent, can also be specified using any of these measures, as well as the printer's measures *ems* and *ens*. These are relative measures, originally based on the size of the letters *m* and *n* in the current type size and typeface. By default, horizontal measures are always taken to be in ems.

There is also a relationship between these units and points and picas. An em is always equivalent in width to the height of the character specified by the point size. In other words, an em in a 12-point type is 12 points wide. An en is always half the size of an em, or half of the current point size. The advantage of using these units is that they are relative to the size of the type being used. This is unimportant in `nroff`, but using these units in `troff` gives increased flexibility to change the appearance of the document without recoding.

The `nroff` and `troff` programs measure not in any of these units, but in device-dependent basic units. Any measures you specify are converted to basic units before they are used. Typically, `nroff` measures in horizontal units of 1/240 of an inch and `otroff` uses a unit of 1/432 inch. These units too are not as arbitrary as they may seem. According to Joseph Osanna's *Nroff/Troff User's Manual*—the original, dense, and authoritative documentation on `troff` published by AT&T as part of the *UNIX Programmer's Manual*—the `nroff` units were chosen as "the least common multiple of the horizontal and vertical resolutions of various typewriter-like output devices." The units for `otroff` were based on the C/A/T typesetter (the device for which `troff` was originally designed), which could move in horizontal increments of 1/432 of an inch and in vertical increments of exactly one-third that, or 1/144 inch. Units for `ditroff` depend on the resolution of the output device. For example, units for a 300 dot-per-inch (dpi) laser printer will be 1/300 of an inch in either a vertical or a horizontal direction. See Appendix D for more information on `ditroff` device units.

You don't need to remember the details of all these measures now. You can generally use the units that are most familiar to you, and we'll come back to the others when we need them.

To specify units, you simply need to add the appropriate scale indicator from Table 4-3 to the numeric value you supply to a formatting request. For example, to space down 3 inches rather than 3 lines, enter the request:

```
.sp 3i
```

The numeric part of any scale indicator can include decimal fractions. Before the specified value is used, `nroff` and `troff` will round the value to the nearest number of device units.

Table 4.3 Units of Measure

Indicator	Units
`c`	Centimeters
`i`	Inches
`m`	Ems
`n`	Ens
`p`	Points
`P`	Picas
`u`	Device Units
`v`	Vertical spaces (lines)
`none`	Default

In fact, you can use any reasonable numeric expression with any request that expects a numeric argument. However, when using arithmetic expressions, you have to be careful about what units you specify. All of the horizontally oriented requests—`.ll`, `.in`, `.ti`, `.ta`, `.po`, `.lt`, and `.mc`—assume you mean ems unless you specify otherwise.

Vertically oriented requests like `.sp` assume `v`'s unless otherwise specified. The only exceptions to this rule are `.ps` and `.vs`, which assume points by default—but these are not really motion requests anyway.

As a result, if you make a request like:

```
.ll 7i/2
```

what you are really requesting is:

```
.ll 7i/2m
```

The request:

```
.ll 7i/2i
```

is not what you want either. In performing arithmetic, as with fractions, the formatter converts scaled values to device units. In `otroff`, this means the previous expression is really evaluated as:

```
.ll (7*432u)/(2*432u)
```

If you really want half of 7 inches, you should specify the expression like this:

```
.ll 7i/2u
```

You could easily divide 7 by 2 yourself and simply specify 3.5i. The point of this example is that when you are doing arithmetic—usually with values stored in variables called number registers (more on these later)—you will need to pay attention to the interaction between units. Furthermore, because fractional device units are always rounded down, you should avoid expressions like `7i/2.5u` because this is equivalent to `7i/2u`.

In addition to absolute values, many `nroff` and `troff` requests allow you to specify relative values, by adding a + or a − before the value. For example:

```
.ll −.5i
```

will subtract ½ inch from the current line length, whatever it is.

Setting Margins

In `nroff` and `troff`, margins are set by the combination of the `.po` (*page offset*) and `.ll` (*line length*) requests. The `.po` request defines the left margin. The `.ll` request defines how long each line will be after filling, and so implicitly defines the right margin:

`po`	`ll`	*right margin*

The `nroff` program's default line length of 6.5 inches is fairly standard for an 8½-by-11 page—it allows for 1-inch margins on either side.

Assuming that we'd like 1¼-inch margins on either side of the page, we would issue the following requests:

```
.ll 6i
.po 1.25i
```

This will give us 1¼ inches for both the right and left margins. The `.po` request specifies a left margin, or page offset, of 1¼ inches. When the 6-inch line length is added to this, it will leave a similar margin on the rlght side of the page.

Let's take a look at how our sample letter will format now. One paragraph of the output should give you the idea.

```
In   our   conversation   last   Thursday,   we
discussed  a  documentation  project  that would
produce a user's guide and reference  manual  for
the  Alcuin  product.  Yesterday,  I received the
product demo and other materials that you sent me.
```

As we saw earlier, `nroff` assumes a default page offset of 0. Either you or the macro package you are using must set the page offset. In `troff`, though, there is a default page offset of 26/27 inch, so you can get away without setting this value.

(Keep in mind that all `nroff` output examples are actually simulated with `troff`, and are reduced to fit on our own 5-inch wide printed page. As a result, the widths shown in our example output are not exact, but are suggestive of what the actual result would be on an 8½-by-11 inch page).

Setting Indents

In addition to the basic page offset, or left margin, you may want to set an indent, either for a single line or an entire block of text. You may also want to center one or more lines of text.

To do a single-line indent, as is commonly used to introduce a paragraph, use the `.ti` (*temporary indent*) request. For example, if you followed the blank lines between paragraphs in the sample letter with the request `.ti 5`, you'd get a result like this from `nroff`:

```
...Yesterday, I received the product demo and other
materials that you sent me.

     Going  through  a  demo  session  gave  me  a
much  better  understanding  of  the  product.   I
confess  to being  amazed by  Alcuin...
```

The `.in` request, by contrast, sets an indent that remains in effect until it is changed. For example, if you had entered the line `.in 5` between the paragraphs, (instead of `.ti 5`), the result would have looked like this:

```
...Yesterday, I received the product demo and other
materials that you sent me.

     Going  through  a  demo  session  gave  me  a
     much better  understanding  of  the  product.
     I confess  to being  amazed by  Alcuin...
```

All succeeding paragraphs will continue to be indented, until the indent is reset. The default indent (the value at the left margin) is 0.

These two indent requests can be combined to give a "hanging indent." Remember that you can specify negative values to many requests that take numeric arguments. Here is the first case where this makes sense. Let's say we would like to modify the letter so that it numbers the points and indents the body of the numbered paragraph:

```
...Yesterday, I received the product demo and other materials
that you sent me.  After studying them, I want to clarify
a couple of points:

.in 4
.ti -4
1.  Going through a demo session gave me a much better
understanding of the product.  I confess to being amazed by
Alcuin...
```

The first line will start at the margin, and subsequent lines will be indented:

```
...Yesterday, I received the product demo and other
materials that you sent me.    After studying them,
I want to clarify a couple of points.

1.  Going through  a demo session  gave  me  a much
    better understanding of the product.  I confess
    to  being  amazed  by  Alcuin...
```

To line up an indented paragraph like this in `nroff`, just count the number of characters you want to space over, then use that number as the size of the indent. But this trick is not so simple in `troff`. Because characters, and even spaces, are not of constant width, it is more difficult to create a hanging indent. Ens are a good unit to use for indents. Like ems, they are relative to the point size, but they are much closer to the average character width than an em. As a result, they are relatively intuitive to work with. An indent of `5n` is about where you expect a 5-character indent to be from familiarity with a typewriter.

Centering Output Lines

Centering is another useful layout tool. To center the next line, use the `.ce` request:

```
.ce
This line will be centered.
```

Here's the result:

```
                    This line will be centered.
```

Centering takes into account any indents that are in effect. That is, if you have used `.in` to specify an indent of 1 inch, and the line length is 5 inches, text will be centered within the 4-inch span following the indent.

To center multiple lines, specify a number as an argument to the request:

```
.ce 3
Documentation for the Alcuin Product

A Proposal Prepared by
Fred Caslon
```

Here's the result:

```
               Documentation for the Alcuin Product

                      A Proposal Prepared by
                           Fred Caslon
```

Notice that `.ce` centered all three *text* lines, ignoring the blank line between.

To center an indeterminately large number of lines, specify a very large number with the `.ce` request, then turn it off by entering `.ce 0`:

```
.ce 1000
Many lines of text here.
.ce 0
```

In looking at the examples, you probably noticed that centering automatically disables filling and justification. Each line is centered individually. However, there is also the case in which you would like to center an entire filled and justified paragraph. (This paragraph style is often used to set off quoted material in a book or paper). You can do this by using both the `.in` and `.ll` requests:

```
I was particularly interested by one comment that I
read in your company literature:
```

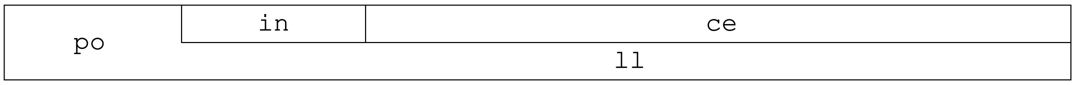
```
.in +5n
.ll −5n
The development of Alcuin can be traced back to our
founder’s early interest in medieval manuscripts.
He spent several years in the seminary before
becoming interested in computers.  After he became
an expert on typesetting software, he resolved to
put his two interests together.
.in −5n
.ll +5n
```

Here’s the result:

```
I was  particularly interested  by  one  comment  that  I
read in your company literature:

    The development of Alcuin can be  traced  back to
    our   founder’s  early  interest    in   medieval
    manuscripts.   He  spent  several  years  in  the
    seminary before becoming  interested  in  comput-
    ers.  After he  became an expert  on  typesetting
    software,  he resolved to put his  two  interests
    together.
```

Remember that a line centered with `.ce` takes into account any indents in effect at the time. You can visualize the relationship between page offset, line length, indents, and centering as follows:



Setting Tabs

No discussion of how to align text would be complete without a discussion of tabs. A tab, as anyone who has used a typewriter well knows, is a horizontal motion to a predefined position on the line.

The problem with using tabs in `nroff` and `troff` is that what you see on the screen is very different from what you get on the page. Unlike a typewriter or a *wysiwyg* word processor, the editor/formatter combination presents you with two different tab settings. You can set tabs in `vi`, and you can set them in `nroff` and `troff`, but the settings are likely to be different, and the results on the screen definitely unaesthetic.

However, after you get used to the fact that tabs will not line up on the screen in the same way as they will on the printed page, you can use tabs quite effectively.

By default, tab stops are set every .8 inches in `nroff` and every .5 inches in `troff`. To set your own tab stops in `nroff` or `troff`, use the `.ta` request. For example:

```
.ta 1i 2.5i 3i
```

will set three tab stops, at 1 inch, 2½ inches, and 3 inches, respectively. Any previous or default settings are now no longer in effect.

You can also set incremental tab stops. The request:

```
.ta 1i +1.5i +.5i
```

will set tabs at the same positions as the previous example. Values preceded with a plus sign are added to the value of the last tab stop.

You can also specify the alignment of text at a tab stop. Settings made with a numeric value alone are left adjusted, just as they are on a typewriter. However, by adding either the letter `R` or `C` to the definition of a tab stop, you can make text right adjusted or centered on the stop.

For example, the following input lines (where a tab character is shown by the symbol |—|):

```
.nf
.ta 1i 2.5i 3.5i
|—|First|—|Second|—|Third
.fi
```

will produce the following output:

```
                First              Second        Third
```

But:

```
.nf
.ta 1i 2.5iR 3.5iC
|—|First|—|Second|—|Third
.fi
```

will produce:

```
                First         Second             Third
```

Right-adjusted tabs can be useful for aligning numeric data. This is especially true in `troff`, where all characters (including blank spaces) have different sizes, and, as a result, you can't just line things up by eye. If the numbers you want to align have an uneven number of decimal positions, you can manually force right adjustment of numeric data using the special escape sequence `\0`, which will produce a blank space exactly the same width as a digit. For example:

```
.ta 1iR
|—|500.2\0
|—|125.35
|—|50.\0\0
```

will produce:

```
          500.2
          125.35
           50.
```

As on a typewriter, if you have already spaced past a tab position (either by printing characters, or with an indent or other horizontal motion), a tab in the input will push text over to the next available tab stop. If you have passed the last tab stop, any tabs present in the input will be ignored.

You must be in no-fill mode for tabs to work correctly. This is not just because filling will override the effect of the tabs. Using `.nf` when specifying tabs is an important rule of thumb; we'll look at the reasoning behind it in Chapter 15.

Underlining

We haven't yet described how to underline text, a primary type of emphasis in `nroff`, which lacks the `troff` ability to switch fonts for emphasis.

There are two underlining requests: `.ul` (*underline*) and `.cu` (*continuous underline*). The `.ul` request underlines only printable characters (the words, but not the spaces), and `.cu` underlines the entire text string.

These requests are used just like `.ce`. Without an argument, they underline the text on the following input line. You can use a numeric argument to specify that more than one line should be underlined.

Both of these requests produce italics instead of underlines in `troff.` Although there is a request, `.uf`, that allows you to reset the underline font to some other font than italics,* there is no way to have these requests produce underlining even in `troff`. (The `ms` and `mm` macro packages both include a *macro* to do underlining in `troff`, but this uses an entirely different mechanism, which is not explained until Chapter 15).

Inserting Vertical Space

As you have seen, a blank line in the input text results in a blank line in the output. You can leave blank space on the page (for example, between the closing of a letter and the signature) by inserting a number of blank lines in the input text.

However, particularly when you are entering formatting codes as you write, rather than going back to code an existing file like our sample letter, it is often more convenient to specify the spacing with the `.sp` request.

For example, you could type:

```
Sincerely,
.sp 3
Fred Caslon
```

In `troff`, the `.sp` request is even more important, because `troff` can space in much finer increments.

For example, if we were formatting the letter with `troff`, a full space between paragraphs would look like this:

In our conversation last Thursday, we discussed a documentation project that would produce a user's guide and reference manual for the Alcuin product. Yesterday, I received the product demo and other materials that you sent me.

Going through a demo session gave me a better understanding of the product. I confess to being amazed by Alcuin. Some people around here, looking over my shoulder, were also astounded by the illuminated manuscript I produced with Alcuin. One person, a student of calligraphy, was really impressed.

---

* This request is generally used when the document is being typeset in a font family other than Times Roman. It might be used to set the "underline font" to Helvetica Italic, rather than the standard Italic.

The output would probably look better if there was a smaller amount of space between the lines. If we replace the line between the paragraphs with the request `.sp .5`, here is what we will get:

> In our conversation last Thursday, we discussed a documentation project that would produce a user's guide and reference manual for the Alcuin product. Yesterday, I received the product demo and other materials that you sent me.
>
> Going through a demo session gave me a much better understanding of the product. I confess to being amazed by Alcuin. Some people around here, looking over my shoulder, were also astounded by the illuminated manuscript I produced with Alcuin. One person, a student of calligraphy, was really impressed.

Although it may not yet be apparent how this will be useful, you can also space to an absolute position on the page, by inserting a vertical bar before the distance. The following:

```
.sp |3i
```

will space down to a position 3 inches from the top of the page, rather than 3 inches from the current position.

You can also use negative values with ordinary relative spacing requests. For example:

```
.sp -3
```

will move back up the page three lines. Of course, when you use any of these requests, you have to know what you are doing. If you tell `nroff` or `troff` to put one line on top of another, that's exactly what you'll get. For example:

```
This is the first line.
.sp -2
This is the second line.
.br
This is the third line.
```

will result in:

```
This is the second line.
This is the first line.
```

Sure enough, the second line is printed above the first, but because we haven't restored the original position, the third line is then printed on top of the first.

When you make negative vertical motions, you should always make compensatory positive motions, so that you end up at the correct position for future output. The previous example would have avoided disaster if it had been coded:

```
This is the first line.
.sp -2
This is the second line.
.sp
This is the third line.
```

(Notice that you need to space down one less line than you have spaced up because, in this case, printing the second line "uses up" one of the spaces you went back on).

These kind of vertical motions are generally used for line drawing (e.g., for drawing boxes around tables), in which all of the text is output, and the formatter then goes back up the page to draw in the lines. At this stage, it is unlikely that you will find an immediate use for this capability. Nonetheless, we are sure that a creative person, knowing that it is there, will find it just the right tool for a job. (We'll show a few creative uses of our own later).

You probably aren't surprised that a typesetter can go back up the page. But you may wonder how a typewriter-like printer can go back up the page like this. The answer is that it can't. If you do any reverse line motions (and you do when you use certain macros in the standard packages, or the `tbl` and `eqn` preprocessors), you must pass the `nroff` output through a special filter program called `col` to get all of the motions sorted out beforehand, so that the page will be printed in the desired order:

```
$ nroff files | col | lp
```

Double or Triple Spacing

Both `nroff` and `troff` provide a request to produce double- or triple-spaced output without individually adjusting the space between each line. For example:

```
.ls 2
```

Putting this at the top of the file produces double-spaced lines. An argument of 3 specifies triple-spaced lines.

▫ Page Transitions ▫

If we want space at the top of our one-page letter, it is easy enough to insert the command:

```
.sp 1i
```

before the first line of the text. However, `nroff` and `troff` do not provide an easy way of handling page transitions in multipage documents.

By default, `nroff` and `troff` assume that the page length is 11 inches. However, neither program makes immediate use of this information. There is no default top and bottom margin, so text output begins on the first line, and goes to the end of the page.

The `.bp` (*break page*) request allows you to force a page break. If you do this, the remainder of the current page will be filled with blank lines, and output will start again at the top of the second page. If you care to test this, insert a `.bp` anywhere in the text of our sample letter, then process the letter with `nroff`. If you save the resulting output in a file:

```
$ nroff letter > letter.out
```

you will find that the text following the `.bp` begins on line 67 (11 inches at 6 lines per inch equals 66 lines per page).

To automatically leave space at the top and bottom of each page, you need to use the `.wh` (*when*) request. In `nroff` and `troff` parlance, this request sets a *trap*—a position on the page at which a given macro will be executed.

You'll notice that we said *macro*, not *request*. There's the rub. To use `.wh`, you need to know how to define a macro. It doesn't work with single requests.

There's not all that much to defining macros, though. A macro is simply a sequence of stored requests that can be executed all at once with a single command. We'll come back to this later, after we've looked at the process of macro definition.

For the moment, let's assume that we've defined two macros, one containing the commands that will handle the top margin, and another for the bottom margin. The first macro will be called `.TM`, and the second `.BM`. (By convention, macros are often given names consisting of uppercase letters, to distinguish them from the basic `nroff` and `troff` requests. However, this is a convention only, and one that is not always followed).

To set traps that will execute these macros, we would use the `.wh` request as follows:

```
.wh 0 TM
.wh −1i BM
```

The first argument to `.wh` specifies the vertical position on the page at which to execute the macro. An argument of 0 always stands for the top of the page, and a negative value is always counted from the bottom of the page, as defined by the page length.

In its simplest form, the `.TM` macro need only contain the single request to space down 1 inch, and `.BM` need only contain the single request to break to a new page. If `.wh` allowed you to specify a single request rather than a macro, this would be equivalent to:

```
.wh 0 .sp 1i
.wh −1i .bp
```

With an 11-inch page length, this would result in an effective 9-inch text area, because on every page, the formatter's first act would be to space down 1 inch, and it would break to a new page when it reached 1 inch from the bottom.

You might wonder why `nroff` and `troff` have made the business of page transition more complicated than any of the other essential page layout tasks. There are two reasons:

- The `nroff` and `troff` programs were designed with the typesetting heritage in mind. Until fairly recently, most typesetters produced continuous output on rolls of photographic paper or film. This output was manually cut and pasted up onto pages.
- Especially in `troff`, page transition is inherently more complex than the other tasks we've described. For example, books often contain headers and footers that are set in different type sizes or styles. At every page transition, the software must automatically save information about the current type style, switch to the style used by the header or footer, and then revert to the original style when it returns to the main text. Or consider the matter of footnotes—the position at which the page ends is different when a footnote is on the page. The page transition trap must make some allowance for this.

In short, what you might like the formatter to do during page transitions can vary. For this reason, the developers of `nroff` and `troff` have allowed users to define their own macros for handling this area.

When you start out with `nroff` or `troff`, we advise you to use one of the ready-made macro packages, `ms` or `mm`. The standard macro package for UNIX systems based on System V is `mm`; the standard on Berkeley UNIX systems is `ms`. Berkeley UNIX systems also support a third macro package called `me`. In addition, there are specialized macro packages for formatting viewgraphs, standard UNIX reference manual pages (`man`), and UNIX permuted indexes (`mptx`). Only the `ms` and `mm` packages are described in this book. The macro packages have already taken into account many of the complexities in page transition (and other advanced formatting problems), and provide many capabilities that would take considerable time and effort to design yourself.

Of course, it is quite possible to design your own macro package, and we will go into all of the details later. (In fact, this book is coded with neither of the standard macro packages, but with one developed by Steve Kochan and Pat Wood of Pipeline Associates, the consulting editors of this series, for use specifically with the Hayden UNIX library).

Page Length Revisited

Before we take a closer look at macros, let's take a moment to make a few more points about page length, page breaks, and the like.

Assuming that some provision has been made for handling page transitions, there are several wrinkles to the requests we have already introduced, plus several new requests that you will probably find useful.

First, let's talk about page length. It's important to remember that the printing area is defined by the interaction of the page length and the location of the traps you define. For example, you could define a text area 7.5 inches high (as we did in preparing copy for this book) either by

- changing the page length to 9.5 inches, and setting 1-inch margins at the top and bottom;
- leaving the page length at 11 inches, and setting 1.75-inch margins at the top and bottom.

In general, we prefer to think of `.pl` as setting the *paper length*, and use the page transition traps to set larger or smaller margins.

However, there are cases where you really are working with a different paper size. A good example of this is printing addresses on envelopes: the physical paper height is about 4 inches (24 lines on a typewriter-like printer printing 6 lines per inch), and we want to print in a narrow window consisting of four or five lines. A good set of definitions for this case would be:

```
.pl 4i
.wh 0 TM
.wh −9v BM
```

with `.TM` containing the request `.sp 9v`, and with `.BM`, as before, containing `.bp`.

There is more to say about traps, but it will make more sense later, so we'll leave the subject for now.

Page Breaks without Line Breaks

Page breaks—we've talked about their use in page transition traps, but they also have a common use on their own. Often, you will want to break a page before it would normally end. For example, if the page breaks right after the first line of a paragraph, you will probably want to force the line onto the next page, rather than leaving an "orphaned" line. Or you might want to leave blank space at the bottom of a page for an illustration. To do this, simply enter a `.bp` at the desired point. A new page will be started immediately.

However, consider the case in which you need to force a break in the middle of a paragraph to prevent a "widowed" line at the top of the next page. If you do this:

```
The medieval masters of calligraphy and illumination
are largely unknown to us.  We thankfully have examples
of their work, and even
.bp
marginal notes by the copyists of some manuscripts,
but the men who produced these minute masterpieces
are anonymous.
```

the `.bp` request will also cause a line break, and the text will not be filled properly:

```
The medieval  masters  of calligraphy  and illumination
are largely unknown to us.  We thankfully have examples
of their work, and even
```

New page begins here

```
marginal notes by the copyists of some manuscripts, but
the  men  who  produced these  minute  masterpieces are
anonymous.
```

Fortunately, there is a way around this problem. If you begin a request with an apostrophe instead of a period, the request will not cause a break.

```
The medieval masters of calligraphy and illumination
are largely unknown to us.  We thankfully have examples
of their work, and even
'bp
marginal notes by the copyists of some manuscripts,
but the men who produced these minute masterpieces
are anonymous.
```

Now we have the desired result:

```
The  medieval  masters of  calligraphy and  illumination
are  largely unknown to us.  We thankfully have examples
```

New page begins here

```
of  their work,  and even marginal notes by the copyists
of  some  manuscripts,  but  the  men who produced these
minute masterpieces are anonymous.
```

(In fact, most page transition macros use this feature to make paragraphs continue across page boundaries. We'll take a closer look at this in later chapters).

Another very useful request is the conditional page break, or `.ne` (*need*) request. If you want to make sure an entire block of text appears on the same page, you can use this request to force a page break if there isn't enough space left. If there is sufficient space, the request is ignored.

For example, the two requests:

```
.ne 3.2i
.sp 3i
```

might be used to reserve blank space to paste in an illustration that is 3 inches high.

The `.ne` request does not cause a break, so you should be sure to precede it with `.br` or another request that causes a break if you don't want the remnants of the current line buffer carried to the next page if the `.ne` is triggered.

It is often better to use `.ne` instead of `.bp`, unless you're absolutely sure that you will *always* want a page break at a particular point. If, in the course of editing, an `.ne` request moves away from the bottom of the page, it will have no effect. But a `.bp` will always start a new page, sometimes leaving a page nearly blank when the text in a file has been changed significantly.

There are other special spacing requests that can be used for this purpose. (Depending on the macro package, these may have to be used). For example, `.sv` (*save space*) requests a block of contiguous space. If the remainder of the page does not contain the requested amount of space, no space is output. Instead, the amount of space requested is remembered and is output when an `.os` (*output saved space*) request is encountered.

These are advanced requests, but you may need to know about them because most macro packages include two other spacing requests in their page transition macros: `.ns` (*no space*) and `.rs` (*restore space*). An `.ns` inhibits the effect of spacing requests; `.rs` restores the effectiveness of such requests.

Both the `ms` and `mm` macros include an `.ns` request in their page transition macros. As a result, if you issue a request like:

```
.sp 3i
```

with 1 inch remaining before the bottom of the page, you will not get 1 inch at the bottom, plus 2 inches at the top of the next page, but only whatever remains at the bottom. The next page will start right at the top.

However, both macro packages also include an `.os` request in their page top macro, so if you truly want 3 inches, use `.sv 3i`, and you will get the expected result.

However, if you use `.sv`, you will also have another unexpected result: text following the spacing request will "float" ahead of it to fill up the remainder of the current page.

We'll talk more about this later. We introduced it now to prevent confusion when spacing requests don't always act the way you expect.

Page Numbering

The `nroff` and `troff` programs keep track of page numbers and make the current page number available to be printed out (usually by a page transition macro). You can artificially set the page number with the `.pn` request:

`.pn 5`	Set the current page number to 5
`.pn +5`	Increment the current page number by 5
`.pn -5`	Decrement the current page number by 5

You can also artificially set the number for the *next* page whenever you issue a `.bp` request, simply by adding a numeric argument:

`.bp 5`	Break the page and set the next page number to 5
`.bp +5`	Break the page and increment the next page number by 5
`.bp -5`	Break the page and decrement the next page number by 5

In addition to inhibiting `.sp`, the `.ns` request inhibits the action of `.bp`, *unless* a page number is specified. This means (at least in the existing macro packages), that the sequence:

```
.bp
.bp
```

will not result in a blank page being output. You will get the same effect as if you had specified only a simple `.bp`. Instead, you should specify:

```
.bp +1
```

The starting page number (usually 1) can also be set from the command line, using the `-n` option. For example:

```
$ nroff -ms -n10 file
```

will start numbering *file* at page number 10. In addition, there is a command-line option to print only selected pages of the output. The `-o` option takes a list of page numbers as its argument. The entire file (up to the last page number in the list) is processed, but only the specified pages are output. The list can include single pages separated by commas, or a range of pages separated by a hyphen, or both. A number followed by a trailing hyphen means to output from that page to the end. For example:

```
$ nroff -ms -o1,5,7-9,13- file
```

will output pages 1, 5, 7 through 9, and from 13 to the end of the file. There should be no spaces anywhere in the list.

▫ Changing Fonts ▫

In old `troff` (`otroff`), you were limited to four fonts at a time, because the fonts had to be physically mounted on the C/A/T typesetter. With `ditroff` and a laser printer or a modern typesetter, you can use a virtually unlimited number of fonts in the same document.

In `otroff` you needed to specify the basic fonts that are in use with the `.fp` (*font position*) request. Normally, at the front of a file (or, more likely, in the macro package), you would use this request to specify which fonts are mounted in each of the four quadrants (positions) of the typesetter wheel. By default, the roman font is mounted in position 1, the italic font in position 2, the bold font in position 3, and the special font in position 4. That is, `troff` acts as though you had included the lines:

```
.fp 1 R
.fp 2 I
.fp 3 B
.fp 4 S
```

In `ditroff`, up to ten fonts are automatically mounted, with the special font in position 10. Which fonts are mounted, and in which positions, depends on the output device. See Appendix D for details. The font that is mounted in position 1 will be used for the body type of the text—it is the font that will be used if no other specification is given. The special font is also used without any intervention on your part when a character not in the normal character set is requested.

To request one of the other fonts, you can use either the `.ft` request, or the inline font-switch escape sequence `\f`.

For example:

```
.ft B
This line will be set in bold type.
.br
.ft R
This line will again be set in roman type.
```

will produce:

This line will be set in bold type.
This line will again be set in roman type.

You can also change fonts using an inline font *escape sequence*. For example, the preceding sentence was coded like this:

```
...an inline font \fIescape sequence\fP.
```

You may wonder at the `\fP` at the end, rather than `\fR`. The `P` command is a special code that can be used with either the `.ft` request or the `\f` escape sequence. It means "return to the previous font, whatever it was." This is often preferable to an explicit font request, because it is more general.

All of this begs the question of fonts different than Times Roman, Bold, and Italic. There are two issues: first, which fonts are available on the output device, and second, which fonts does `troff` have width tables for. (As described previously, `troff` uses these tables to determine how far to space over after it outputs each character). For `otroff` these width tables are in the directory `/usr/lib/font`, in files whose names begin with `ft`. If you list the contents of this directory, you might see something like this for `otroff`:

```
$ ls /usr/lib/font
ftB   ftBC  ftC   ftCE  ftCI
ftCK  ftCS  ftCW  ftFD  ftG
ftGI  ftGM  ftGR  ftH   ftHB
ftHI  ftI   ftL   ftLI  ftPA
ftPB  ftPI  ftR   ftS   ftSB
ftSI  ftSM  ftUD
```

You can pick out the familiar R, I, B, and S fonts, and may guess that `ftH`, `ftHI`, and `ftHB` refer to Helvetica, Helvetica Italic, and Helvetica Bold fonts. However, unless you are familiar with typesetting, the other names might as well be Greek to you. In any event, these width tables, normally supplied with `troff`, are for fonts that are commonly used with the C/A/T typesetter. *If you are using a different device, they may be of no use to you.*

The point is that if you are using a different typesetting device, you will need to get information about the font names for your system from whoever set up the equipment to work with `troff`. The contents of `/usr/lib/font` will vary from installation to installation, depending on what fonts are supported.

For `ditroff`, there is a separate subdirectory in `/usr/lib/font` for each supported output device. For example:

```
$ ls /usr/lib/font
devlj    devps
$ ls /usr/lib/font/devps
B.out    BI.out    CB.out    CI.out    CW.out    CX.out
DESC.out H.out     HB.out    HI.out    HK.out    HO.out
HX.out   I.out     LI.out    PA.out    PB.out    PI.out
PX.out   R.out     O.out     RS.out    S.out     S1.out
```

Here, the font name is followed by the string `.out`.

Again, the font names themselves are probably Greek to you. However, with `ditroff`, you can actually use any of these names, and see what results they give you, because all fonts should be available at any time.

For the sake of argument, let's assume that your typesetter or other `troff`-compatible equipment supports the Helvetica font family shown in Figure 4-3, with the names `H`, `HI`, and `HB`. (This is a fairly reasonable assumption, because Helvetica is probably the most widely available font family after Times).

When specifying two-character font names with the `\f` escape sequence, you must add the `(` prefix as well. For example, you would specify Helvetica Italic by the inline sequence `\f(HI`, and Helvetica Bold by `\f(HB`.

There is another issue when you are using fonts other than the Times Roman family. Assume that you decide to typeset your document in Helvetica rather than Roman. You reset your initial font position settings to read:

```
.fp 1 H
.fp 2 HI
.fp 3 HB
.fp 4 S
```

However, throughout the text, you have requests of the form:

```
.ft B
```

or:

```
\fB
```

You will need to make a set of global replacements throughout your file. To insulate yourself in a broader way from overall font change decisions, `troff` allows you to specify fonts by position, even within `.ft` and `\f` requests:

`.ft 1`	or	`\f1`	Use the font mounted in position 1
`.ft 2`	or	`\f2`	Use the font mounted in position 2
`.ft 3`	or	`\f3`	Use the font mounted in position 3
`.ft 4`	or	`\f4`	Use the font mounted in position 4

Because you don't need to use the `.fp` request to set font positions with `ditroff`, and the range of fonts is much greater, you may have a problem knowing which fonts are mounted in which positions. A quick way to find out which fonts are mounted is to run `ditroff` on a short file, sending the output to the screen. For example:

```
$ ditroff -Tps junk | more
x T ps
x res 720 1 1
x init
x font 1 R
x font 2 I
x font 3 B
x font 4 BI
x font 5 CW
```

Helvetica

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + - . , / : ; = ? [ ] |
• □ — – _ ¼ ½ ¾ fi fl ° † ´ ® ©

Helvetica Italic

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + - . , / : ; = ? [ ] |
• □ — – _ ¼ ½ ¾ fi fl ° † ´ ® ©

Helvetica Bold

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
1234567890
! $ % & ( ) ‘ ’ * + - . , / : ; = ? [ ] |
• □ — – _ ¼ ½ ¾ fi fl ° † ´ ® ©

Special Mathematical Font

" ´ \ ^ _ ` ~ / 〈 〉 { } # @ + − = ∗
α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ σ ς τ υ ϕ χ ψ ω
Γ Δ Θ Λ Ξ Π Σ ϒ Φ Ψ Ω
√ ‾ ≥ ≤ ≡ ~ ≠ → ← ↑ ↓ × ÷ ± ∪ ∩ ⊂ ⊃ ⊆ ⊇ ∞ ∂
§ ∇ ¬ ∫ ∝ ∅ ∈ ‡ ☞ ☜ | ○ ⎧ ⎩ ⎫ ⎭ ⎨ ⎬ ⎪ ⎣ ⎦ ⎡ ⎤ ‖

Figure 4.3 Helvetica Fonts

```
x font 6 CB
x font 7 H
x font 8 HB
x font 9 HI
x font 10 S
...
```

The font positions should appear at the top of the file. In this example, you see the following fonts: (Times) Roman, (Times) Bold, (Times) Italic, (Times) Bold Italic, Constant Width, Constant Bold, Helvetica, Helvetica Bold, Helvetica Italic, and Special. Which font is mounted in which position is controlled by the file `DESC.out` in the device subdirectory of `/usr/lib/font`. See Appendix D for details.

Special Characters

A variety of special characters that are not part of the standard ASCII character set are supported by `nroff` and `troff`. These include Greek letters, mathematical symbols, and graphic characters. Some of these characters are part of the font referred to earlier as the *special font*. Others are part of the standard typesetter fonts.

Regardless of the font in which they are contained, special characters are included in a file by means of special four-character escape sequences beginning with `\(`.

Appendix B gives a complete list of special characters. However, some of the most useful are listed in Table 4-4, because even as a beginner you may want to include them in your text. Although `nroff` makes a valiant effort to produce some of these characters, they are really best suited for `troff`.

Table 4.4 Special Characters

Name	Escape Sequence	Output Character
em dash	\(em	—
bullet	\(bu	•
square	\(sq	□
baseline rule	\(ru	_
underrule	\(ul	_
1/4	\(14	¼
1/2	\(12	½
3/4	\(34	¾
degrees	\(de	°
dagger	\(dg	†
double dagger	\(dd	‡
registered mark	\(rg	®
copyright symbol	\(co	©
section mark	\(sc	§
square root	\(sr	√
greater than or equal	\(>=	≥
less than or equal	\(<=	≤
not equal	\(!=	≠
multiply	\(mu	×
divide	\(di	÷
plus or minus	\(+-	±
right arrow	\(->	→
left arrow	\(<-	←
up arrow	\(ua	↑
down arrow	\(da	↓

We'll talk more about some of these special characters as we use them. Some are used internally by `eqn` for producing mathematical equations. The use of symbols such as the copyright, registered trademark, and dagger is fairly obvious.

However, you shouldn't limit yourself to the obvious. Many of these special characters can be put to innovative use. For example, the square root symbol can be used to simulate a check mark, and the square can become an alternate type of bullet. As we'll show in Chapter 15, you can create additional, effective character combinations, such as a checkmark in a box, with overstriking.

The point is to add these symbols to your repertoire, where they can wait until need and imagination provide a use for them.

Type Size Specification

Typesetting also allows for different overall sizes of characters. Typesetting character sizes are described by units called *points.* A point is approximately 1/72 of an inch. Typical type sizes range from 6 to 72 points. A few different sizes follow:

This line is set in 6-point type.

This line is set in 8-point type.

This line is set in 10-point type.

This line is set in 12-point type.

This line is set in 14-point type.

This line is set in 18-point type.

(The exact size of a typeface does not always match its official size designation. For example, 12-point type is not always 1/6 inch high, nor is 72-point type 1 inch high. The precise size will vary with the typeface).

As with font changes, there are two ways to make size changes: with a request and with an inline escape sequence. The `.ps` request sets the point size. For example:

`.ps 10`   Set the point size to 10 points

A `.ps` request that does not specify any point size reverts to the previous point size setting, whatever it was:

`.ps 10`

Some text here

`.ps`   Revert to the point size before we changed it

To switch point size in the middle of the line, use the `\s` escape sequence. For example, many books reduce the point size when they print the word UNIX in the middle of a line. The preceding sentence was produced by these input lines:

```
For example, many books reduce the point size when
they print the word \s8UNIX\s0 in the middle of a line.
```

As you can probably guess from the example, `\s0` does not mean to use a point size of 0, but to revert to the previous size.

In addition, you can use relative values when specifying point sizes. Knowing that the body of the book is set in 10-point type, we could have achieved the same result by entering:

```
For example, many books reduce the point size when
they print the word \s-2UNIX\s0 in the middle of a line.
```

You can increment or decrement point sizes only using a single digit; that is, you can't increment or decrement the size by more than 9 points.

Only certain sizes may be available on the typesetter. (Legal point sizes in `otroff` are 6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 28, and 36. Legal point sizes in `ditroff` depend upon the output device, but there will generally be more sizes available). If you request a point size between two legal sizes, `otroff` will round up to the next legal point size; `ditroff` will round to the nearest available size.

Vertical Spacing

In addition to its ability to change typefaces and type sizes on the same page, a typesetter allows you to change the amount of vertical space between lines. This spacing is sometimes referred to as the *baseline spacing* because it is the distance between the base of characters on successive lines. (The difference between the point size and the baseline spacing is referred to as *leading*, from the old days when a human compositor inserted thin strips of lead between successive lines of type).

A typewriter or typewriter-style printer usually spaces vertically in 1/6-inch increments (i.e., 6 lines per inch). A typesetter usually adjusts the space according to the point size. For example, the type samples shown previously were all set with 20 points of vertical space. More typically, the vertical space will vary along with the type size, like this:

This line is set in 6-point type and 8-point spacing.
This line is set in 8-point type and 10-point spacing.
This line is set in 10-point type and 12-point spacing.
This line is set in 12-point type and 14-point spacing.
This line is set in 14-point type and 16-point spacing.
This line is set in 18-point type and 20-poi

Typically, the body of a book is set with a single size of type (usually 9 or 10 point, with vertical spacing set to 11 or 12 points, respectively). Larger sizes are used occasionally for emphasis, for example, in chapter or section headings. When the type size is changed, the vertical spacing needs to be changed too, or the type will overrun the previous line, as follows, where 14-point type is shown with only 10-point spacing.

Here is type larger than
the space allotted for it.

Vertical spacing is changed with the `.vs` request. A vertical space request will typically be paired with a point size request:

```
.ps 10
.vs 12
```

After you set the vertical spacing with `.vs`, this becomes the basis of the `v` unit for `troff`. For example, if you enter `.vs 12`, the request `.sp` will space down 12 points; the request:

```
.sp 0.5v
```

will space down 6 points, or half the current vertical line spacing. However, if you change the baseline vertical spacing to 16, the `.sp` request will space down 16 points. Spacing specified in any other units will be unaffected. What all this adds up to is the commonsense observation that a blank line takes up the same amount of space as one containing text.

When you use double and triple spacing, it applies a multiplication factor to the baseline spacing. The request `.ls 2` will double the baseline spacing. You can specify any multiplication factor you like, though 2 and 3 are the most reasonable values.

The `.ls` request will only affect the spacing between output lines of text. It does not change the definition of `v` or affect vertical spacing requests.

▫ A First Look at Macros ▫

Although we won't go into all the details of macro design until we have discussed the existing macro packages in the next two chapters, we'll cover some of the basic concepts here. This will help you understand what the macro packages are doing and how they work.

To define a macro, you use the `.de` request, followed by the sequence of requests that you want to execute when the macro is invoked. The macro definition is terminated by the request `..` (two dots). The name to be assigned to the macro is given as an argument to the `.de` request.

You should consider defining a macro whenever you find yourself issuing a repetitive sequence of requests. If you are not using one of the existing macro packages (which have already taken care of this kind of thing), paragraphing is a good example of the kind of formatting that lends itself to macros.

Although it is certainly adequate to separate paragraphs simply by a blank line, you might instead want to separate them with a blank line and a temporary indent. What's more, to prevent "orphaned" lines, you would like to be sure that at least two lines of each paragraph appear at the bottom of the page. So you might define the following macro:

```
.de P
.sp
.ne 2
.ti 5n
..
```

This is the simplest kind of macro—a straightforward sequence of stored commands. However, macros can take arguments, take different actions depending on the presence or absence of various conditions, and do many other interesting and wonderful things.

We'll talk more about the enormous range of potential in macros in later chapters. For the moment, let's just consider one or two points that you will need to understand in order to use the existing macro packages.

Macro Arguments

Most basic `troff` requests take simple arguments—single characters or letters. Many macros take more complex arguments, such as character strings. There are a few simple pointers you need to keep in mind through the discussion of macro packages in the next two chapters.

First, a space is taken by default as the separator between arguments. If a single macro argument is a string that contains spaces, you need to quote the entire string to keep it from being treated as a series of separate arguments.

For example, imagine a macro to print the title of a chapter in this book. The macro call looks like this:

```
.CH 4 "Nroff and Troff"
```

A second point: to skip an argument that you want to ignore, supply a null string (`""`). For example:

```
.CH ""  "Preface"
```

As you can see, it does no harm to quote a string argument that doesn't contain spaces (`"Preface"`), and it is probably a good habit to quote all strings.

Number Registers

When you use a specific value in a macro definition, you are limited to that value when you use the macro. For example, in the paragraph macro definition shown previously, the space will always be 1, and the indent always `5n`.

However, `nroff` and `troff` allow you to save numeric values in special variables known as *number registers*. If you use the value of a register in a macro definition, the action of the macro can be changed just by placing a new value in the register. For example, in `ms`, the size of the top and bottom margins is not specified with an absolute value, but with a number register. As a result, you don't need to change the macro definition to change these margins; you simply reset the value of the appropriate number register. Just as importantly, the contents of number registers can be used as *flags* (a kind of message between macros). There are conditional statements in the markup language of `nroff` and `troff`, so that a macro can say: "If number register *Y* has the value *x*, then do thus-and-so. Otherwise, do this." For example, in the `mm` macros, hyphenation is turned off by default. To turn it on, you set the value of a certain number register to 1. Various macros test the value of this register, and use it as a signal to re-enable hyphenation.

To store a value into a number register, use the `.nr` request. This request takes two arguments: the name of a number register,* and the value to be placed into it.

For example, in the `ms` macros, the size of the top and bottom margins is stored in the registers `HM` (*header margin*) and `FM` (*footer margin*). To reset these margins from their default value of 1 inch to 1.75 inches (thus producing a shorter page like the one used in this book), all you would need to do is to issue the requests:

```
.nr HM 1.75i
.nr FM 1.75i
```

You can also set number registers with single-character names from the command line by using the `-r` option. (The `mm` macros make heavy use of this capability). For example:

```
$ nroff -mm -rN1 file
```

will format *file* using the `mm` macros, with number register `N` set to the value `1`. We will talk more about using number registers later, when we describe how to write your own macros. For the moment, all you need to know is how to put new values into existing registers. The next two chapters will describe the particular number registers that you may find useful with the `mm` and `ms` macro packages.

Predefined Strings

The `mm` and `ms` macro packages also make use of some predefined text strings. The `nroff` and `troff` programs allow you to associate a text string with a one- or two-character string name. When the formatter encounters a special escape sequence including the string name, the complete string is substituted in the output.

To define a string, use the `.ds` request. This request takes two arguments, the string name and the string itself. For example:

```
.ds nt Nroff and Troff
```

The string should *not* be quoted. It can optionally begin with a quotation mark, but it should not end with one, or the concluding quotation mark will appear in the output. If you want to *start* a string with one or more blank spaces, though, you should begin the definition with a quotation mark. Even in this case, there is no concluding quotation mark. As always, the string is terminated by a newline.

You can define a multiline string by hiding the newlines with a backslash. For example:

```
.ds LS This is a very long string that goes over \
```

*Number register names can consist of either one or two characters, just like macro names. However, they are distinct—that is, a number register and a macro can be given the same name without conflict.

```
more than one line.
```

When the string is interpolated, it will be subject to filling (unless no-fill mode is in effect) and may not be broken into lines at the same points as you've specified in the definition. To interpolate the string in the output, you use one of the following escape sequences:

```
\*a
\*(ab
```

where *a* is a one-character string name, and *ab* is a two-character string name.

To use the *nt* string we defined earlier, you would type:

```
\*(nt
```

It would be replaced in the output by the words *Nroff and Troff*.

Strings use the same pool of names as macros. Defining a string with the same name as an existing macro will make the macro inoperable, so it is not advisable to go around wildly defining shorthand strings. The `vi` editor's abbreviation facility (described in Chapter 7) is a more effective way to save yourself work typing.

Strings are useful in macro design in much the same way number registers are—they allow a macro to be defined in a more general way. For example, consider this book, which prints the title of the chapter in the header on each odd-numbered page. The chapter title is not coded into the page top macro. Instead, a predefined string is interpolated there. The same macro that describes the format of the chapter title on the first page of the chapter also defines the string that will appear in the header.

In using each of the existing macro packages, you may be asked to define or interpolate the contents of an existing string. For the most part, though, string definitions are hidden inside macro definitions, so you may not run across them. However, there are a couple of handy predefined strings you may find yourself using, such as:

```
\*(DY
```

which always contains the current date in the `ms` macro package. (The equivalent string in `mm` is `\*(DT`). For example, if you wanted a form letter to contain the date that it was formatted and printed rather than the date it was written, you could interpolate this string.

Just What Is a Macro Package?

Before leaving the topic of macros, we ought to take a moment to treat a subject we have skirted up to this point: just what is a macro package?

As the name suggests, a macro package is simply a collection of macro definitions. The fact that there are command-line options for using the existing packages may seem to give them a special status, but they are text files that you can read and modify (assuming that your system has the UNIX file permissions set up so you can do so).

There is no magic to the options `−ms` and `−mm`. The actual option to `nroff` and `troff` is `−m`*x*, which tells the program to look in the directory `/usr/lib/tmac` for a file with a name of the form `tmac.`*x*. As you might expect, this means that there is a file in that directory called `tmac.s` or `tmac.m` (depending on which package you have on your system). It also means that you can invoke a macro package of your own from the command line simply by storing the macro definitions in a file with the appropriate pathname. This file will be added to any other files in the formatting run. This means that if you are using the `ms` macros you could achieve the same result by including the line:

```
.so /usr/lib/tmac/tmac.s
```

at the start of each source file, and omitting the command-line switch `−ms`. (The `.so` request reads another file into the input stream, and when its contents have been exhausted, returns to the current file. Multiple `.so` requests can be nested, not just to read in macro definitions, but also to read in additional text files).

The macros in the standard macro packages are no different (other than in complexity) than the macros you might write yourself. In fact, you can print out and study the contents of the existing macro packages to learn how they work. We'll be looking in detail at the actions of the existing macro packages, but for copyright reasons we can't actually show their internal design. We'll come back to all this later. For now, all you need to know is that macros aren't magic—just an assemblage of simple commands working together.