A Tutorial Introduction to the UNIX Text Editor

Brian W. Kernighan

ABSTRACT

Almost all text input on the UNIX† operating system is done with the text-editor This memorandum is a tutorial guide to help beginners get started with text editing.

Although it does not cover everything, it does discuss enough for most users' day-to-day needs. This includes printing, appending, changing, deleting, moving and inserting entire lines of text; reading and writing files; context searching and line addressing; the substitute command; the global commands; and the use of special characters for advanced editing.

August 3, 1987

---

† UNIX is a trademark of Bell Laboratories.

A Tutorial Introduction to the UNIX Text Editor

Brian W. Kernighan

Introduction

Ed is a ‘‘text editor’’, that is, an interactive program for creating and modifying ‘‘text’’, using directions provided by a user at a terminal. The text is often a document like this one, or a program or perhaps data for a program.

This introduction is meant to simplify learning *ed.* The recommended way to learn *ed* is to read this document, simultaneously using *ed* to follow the examples, then to read the description in section I of the *Programmer’s Manual,* all the while experimenting with *ed.* (Solicitation of advice from experienced users is also useful.)

Do the exercises! They cover material not completely discussed in the actual text. An appendix summarizes the commands.

Disclaimer

This is an introduction and a tutorial. For this reason, no attempt is made to cover more than a part of the facilities that *ed* offers (although this fraction includes the most useful and frequently used parts). When you have mastered the Tutorial, try *Advanced Editing on Also,* there is not enough space to explain basic procedures. We will assume that you know how to log on to and that you have at least a vague understanding of what a file is. For more on that, read *for Beginners.*

You must also know what character to type as the end-of-line on your particular terminal. This character is the key on most terminals. Throughout, we will refer to this character, whatever it is, as

Getting Started

We’ll assume that you have logged in to your system and it has just printed the prompt character, usually either a $ or a %. The easiest way to get *ed* is to type ed (followed by a return) You are now ready to go – *ed* is waiting for you to tell it what to do.

Creating Text – the Append command ‘‘a’’

As your first problem, suppose you want to create some text starting from scratch. Perhaps you are typing the very first draft of a paper; clearly it will have to start somewhere, and undergo modifications later. This section will show how to get some text in, just to get started. Later we’ll talk about how to change it.

When *ed* is first started, it is rather like working with a blank piece of paper – there is no text or information present. This must be supplied by the person using *ed;* it is usually done by typing in the text, or by reading it into *ed* from a file. We will start by typing in some text, and return shortly to how to read files.

First a bit of terminology. In *ed* jargon, the text being worked on is said to be ‘‘kept in a buffer.’’ Think of the buffer as a work space, if you like, or simply as the information that you are going to be editing. In effect the buffer is like the piece of paper, on which we will write things, then change some of them, and finally file the whole thing away for another day.

The user tells *ed* what to do to his text by typing instructions called ‘‘commands.’’ Most commands consist of a single letter, which must be typed in lower case. Each command is typed on a separate line. (Sometimes the command is preceded by information about what line or lines of text are to be affected – we will discuss these shortly.) *Ed* makes no response to most commands – there is no prompting or typing of messages like ‘‘ready’’. (This silence is preferred by experienced users, but sometimes a hangup for beginners.)

The first command is *append,* written as the letter a all by itself. It means ‘‘append (or add) text lines to the buffer, as I type them in.’’ Appending is rather like writing fresh material on a piece of paper.

So to enter lines of text into the buffer, just type an a followed by a followed by the lines of text you want, like this: a Now is the time for all good men to come to the aid of their party.

The only way to stop appending is to type a line that contains only a period. The ‘‘’’ is used to tell *ed* that you have finished appending. (Even experienced users forget that terminating ‘‘’’ sometimes. If *ed* seems to be ignoring you, type an extra line with just ‘‘’’ on it. You may then find you’ve added some garbage lines to your text, which you’ll have to take out later.)

After the append command has been done, the buffer will contain the three lines Now is the time for all good men to come to the aid of their party. The a’’ and ‘‘’’ aren’t there, because they are not text.

To add more text to what you already have, just issue another a command, and continue typing.

Error Messages – ‘‘?’’

If at any time you make an error in the commands you type to *ed,* it will tell you by typing ? This is about as cryptic as it can be, but with practice, you can usually figure out how you goofed.

Writing text out as a file – the Write command ‘‘w’’

It’s likely that you’ll want to save your text for later use. To write out the contents of the buffer onto a file, use the *write* command w followed by the filename you want to write on. This will copy the buffer’s contents onto the specified file (destroying any previous information on the file). To save the text on a file named junk, for example, type w junk Leave a space between w and the file name. *Ed* will respond by printing the number of characters it wrote out. In this case, *ed* would respond with 68 (Remember that blanks and the return character at the end of each line are included in the character count.) Writing a file just makes a copy of the text – the buffer’s contents are not disturbed, so you can go on adding lines to it. This is an important point. *Ed* at all times works on a copy of a file, not the file itself. No change in the contents of a file takes place until you give a w command. (Writing out the text onto a file from time to time as it is being created is a good idea, since if the system crashes or if you make some horrible mistake, you will lose all the text in the buffer but any text that was written onto a file is relatively safe.)

Leaving ed – the Quit command ‘‘q’’

To terminate a session with save the text you’re working on by writing it onto a file using the w command, and then type the command q which stands for The system will respond with the prompt character $ or %). At this point your buffer vanishes, with all its text, which is why you want to write it out before quitting.†

Exercise 1:

Enter *ed* and create some text using a . . . text . . . Write it out using w. Then leave *ed* with the q command, and print the file, to see that everything worked. (To print a file, say pr filename or cat filename in response to the prompt character. Try both.)

Reading text from a file – the Edit command ‘‘e’’

A common way to get text into the buffer is to read it from a file in the file system. This is what you do to edit text that you saved with the w command in a previous session. The *edit* command e fetches the entire contents of a file into the buffer. So if you had saved the three lines ‘‘Now is the time’’, etc., with a w command in an earlier session, the *ed* command e junk would fetch the entire contents of the file junk into the buffer, and respond 68 which is the number of characters in junk. *If anything was already in the buffer, it is deleted first.*

If you use the e command to read a file into the buffer, then you need not use a file name after a subsequent w command; *ed* remembers the last file name used in an e command, and w will write on this file. Thus a good way to operate is ed e file [editing session] w q This way, you can simply say w from time to time, and be secure in the knowledge that if you got the file name right at the beginning, you are writing into the proper file each time.

You can find out at any time what file name *ed* is remembering by typing the *file* command f. In this example, if you typed f *ed* would reply junk

Reading text from a file – the Read command ‘‘r’’

Sometimes you want to read a file into the buffer without destroying anything that is already there. This is done by the *read* command r. The command r junk will read the file junk into the buffer; it adds it to the end of whatever is already in the buffer. So if you do a read after an edit: e junk r junk the buffer will contain *two* copies of the text (six lines). Now is the time for all good men to come to the aid of their party. Now is the time for all good men to come to the aid of their party. Like the w and e commands, r prints the number of characters read in, after the reading operation is complete.

Generally speaking, r is much less used than e.

Exercise 2:

Experiment with the e command – try reading and printing various files. You may get an error ?name, where name is the name of a file; this means that the file doesn’t exist, typically because you spelled the file name wrong, or perhaps that you are not allowed to read or write it. Try alternately reading and appending to see that they work similarly. Verify that ed filename is exactly equivalent to ed e filename What does f filename do?

Printing the contents of the buffer – the Print command ‘‘p’’

To *print* or list the contents of the buffer (or parts of it) on the terminal, use the print command p The way this is done is as follows. Specify the lines where you want printing to begin and where you want it to end, separated by a comma, and followed by the letter p. Thus to print the first two lines of the buffer, for example, (that is, lines 1 through 2) say 1,2p (starting line=1, ending line=2 p) *Ed* will respond with Now is the time for all good men

---

† Actually, will print ? if you try to quit without writing. At that point, write if you want; if not, another q will get you out regardless.

Suppose you want to print *all* the lines in the buffer. You could use 1,3p as above if you knew there were exactly 3 lines in the buffer. But in general, you don't know how many there are, so what do you use for the ending line number? *Ed* provides a shorthand symbol for ''line number of last line in buffer'' − the dollar sign $. Use it this way: 1,$p This will print *all* the lines in the buffer (line 1 to last line.) If you want to stop the printing before it is finished, push the or Delete key; *ed* will type ? and wait for the next command.

To print the *last* line of the buffer, you could use $,$p but *ed* lets you abbreviate this to $p You can print any single line by typing the line number followed by a p. Thus 1p produces the response Now is the time which is the first line of the buffer.

In fact, *ed* lets you abbreviate even further: you can print any single line by typing *just* the line number − no need to type the letter p. So if you say $ *ed* will print the last line of the buffer.

You can also use $ in combinations like $−1,$p which prints the last two lines of the buffer. This helps when you want to see how far you got in typing.

Exercise 3:

As before, create some text using the a command and experiment with the p command. You will find, for example, that you can't print line 0 or a line beyond the end of the buffer, and that attempts to print a buffer in reverse order by saying 3,1p don't work.

The current line − ''Dot'' or ''.''

Suppose your buffer still contains the six lines as above, that you have just typed 1,3p and *ed* has printed the three lines for you. Try typing just p (no line numbers) This will print to come to the aid of their party. which is the third line of the buffer. In fact it is the last (most recent) line that you have done anything with. (You just printed it!) You can repeat this p command without line numbers, and it will continue to print line 3.

The reason is that *ed* maintains a record of the last line that you did anything to (in this case, line 3, which you just printed) so that it can be used instead of an explicit line number. This most recent line is referred to by the shorthand symbol (pronounced ''dot''). Dot is a line number in the same way that $ is; it means exactly ''the current line'', or loosely, ''the line you most recently did something to.'' You can use it in several ways − one possibility is to say ,$p This will print all the lines from (including) the current line to the end of the buffer. In our example these are lines 3 through 6.

Some commands change the value of dot, while others do not. The p command sets dot to the number of the last line printed; the last command will set both

and $ to 6.

Dot is most useful when used in combinations like this one: +1 (or equivalently, +1p) This means ''print the next line'' and is a handy way to step slowly through a buffer. You can also say −1 (or −1p) which means ''print the line *before* the current line.'' This enables you to go backwards if you wish. Another useful one is something like −3,−1p which prints the previous three lines.

Don't forget that all of these change the value of dot. You can find out what dot is at any time by typing = *Ed* will respond by printing the value of dot.

Let's summarize some things about the p command and dot. Essentially p can be preceded by 0, 1, or 2 line numbers. If there is no line number given, it prints the ''current line'', the line that dot refers to. If there is one line number given (with or without the letter p), it prints that line (and dot is set there); and if there are two line numbers, it prints all the lines in that range (and sets dot to the last line printed.) If two line numbers are specified the first can't be bigger than the second (see Exercise 2.)

Typing a single return will cause printing of the next line − it's equivalent to Try it. Try typing a −; you will find that it's equivalent to

Deleting lines: the ''d'' command

Suppose you want to get rid of the three extra lines in the buffer. This is done by the *delete* command d Except that d deletes lines instead of printing them, its action is similar to that of p. The lines to be deleted are specified for d exactly as they are for p: *starting line, ending line* d Thus the command 4,$d deletes lines 4 through the end. There are now three lines left, as you can check by using 1,$p And notice that $ now is line 3! Dot is set to the next line after the last line deleted, unless the last line deleted is the last line in the buffer. In that case, dot is set to $.

Exercise 4:

Experiment with a, e, r, w, p and d until you are sure that you know what they do, and until you understand how dot, $, and line numbers are used.

If you are adventurous, try using line numbers with a, r and w as well. You will find that a will append lines *after* the line number that you specify (rather than after dot); that r reads a file in *after* the line number you specify (not necessarily at the end of the buffer); and that w will write out exactly the lines you specify, not necessarily the whole buffer. These variations are sometimes handy. For instance you can insert a file at the beginning of a buffer by saying 0r filename and you can enter lines at the beginning of the buffer by saying 0a . . . *text* . . .

Notice that is *very* different from

w

Modifying text: the Substitute command ‘‘s’’

We are now ready to try one of the most important of all commands − the substitute command s This is the command that is used to change individual words or letters within a line or group of lines. It is what you use, for example, for correcting spelling mistakes and typing errors.

Suppose that by a typing error, line 1 says Now is th time − the has been left off You can use s to fix this up as follows: 1s/th/the/ This says: ‘‘in line 1, substitute for the characters the characters To verify that it works will not print the result automatically) say p and get Now is the time which is what you wanted. Notice that dot must have been set to the line where the substitution took place, since the p command printed that line. Dot is always set this way with the s command.

The general way to use the substitute command is *starting-line, ending-line* s/*change this*/*to this*/ Whatever string of characters is between the first pair of slashes is replaced by whatever is between the second pair, in *all* the lines between *starting-line* and *ending-line.* Only the first occurrence on each line is changed, however. If you want to change *every* occurrence, see Exercise 5. The rules for line numbers are the same as those for p, except that dot is set to the last line changed. (But there is a trap for the unwary: if no substitution took place, dot is *not* changed. This causes an error ? as a warning.)

Thus you can say 1,$s/speling/spelling/ and correct the first spelling mistake on each line in the text. (This is useful for people who are consistent misspellers!)

If no line numbers are given, the s command assumes we mean ‘‘make the substitution on line dot’’, so it changes things only on the current line. This leads to the very common sequence s/something/something else/p which makes some correction on the current line, and then prints it, to make sure it worked out right. If it didn’t, you can try again. (Notice that there is a p on the same line as the s command. With few exceptions, p can follow any command; no other multi-command lines are legal.)

It’s also legal to say s/ . . . // which means ‘‘change the first string of characters to i.e., remove them. This is useful for deleting extra words in a line or removing extra letters from words. For instance, if you had Nowxx is the time you can say s/xx//p to get Now is the time Notice that // (two adjacent slashes) means ‘‘no characters’’, not a blank. There *is* a difference! (See below for another meaning of //.)

Exercise 5:

Experiment with the substitute command. See what happens if you substitute for some word on a line with several occurrences of that word. For example, do this: a the other side of the coin s/the/on the/p You will get on the other side of the coin A substitute command changes only the first occurrence of the first string. You can change all occurrences by adding a g (for ‘‘global’’) to the s command, like this: s/ . . . / . . . /gp Try other characters instead of slashes to delimit the two sets of characters in the s command − anything should work except blanks or tabs.

(If you get funny results using any of the characters ˆ $ [* \ & read the section on ‘‘Special Characters’’.)

Context searching − ‘‘/ . . . /’’

With the substitute command mastered, you can move on to another highly important idea of *ed* − context searching.

Suppose you have the original three line text in the buffer: Now is the time for all good men to come to the aid of their party. Suppose you want to find the line that contains so you can change it to Now with only three lines in the buffer, it’s pretty easy to keep track of what line the word is on. But if the buffer contained several hundred lines, and you’d been making changes, deleting and rearranging lines, and so on, you would no longer really know what this line number would be. Context searching is simply a method of specifying the desired line, regardless of what its number is, by specifying some context on it.

The way to say ‘‘search for a line that contains this particular string of characters’’ is to type /*string of characters we want to find*/ For example, the *ed* command /their/ is a context search which is sufficient to find the desired line − it will locate the next occurrence of the characters between slashes (‘‘their’’). It also sets dot to that line and prints the line for verification: to come to the aid of their party. ‘‘Next occurrence’’ means that *ed* starts looking for the string at line searches to the end of the buffer, then continues at line 1 and searches to line dot. (That is, the search ‘‘wraps around’’ from $ to 1.) It scans all the lines in the buffer until it either finds the desired line or gets back to dot again. If the given string of characters can’t be found in any line, *ed* types the error message ? Otherwise it prints the line it found.

You can do both the search for the desired line *and* a substitution all at once, like this: /their/s/their/the/p which will yield to come to the aid of the party. There were three parts to that last command: context search for the desired line, make the substitution, print the line.

The expression /their/ is a context search expression. In their simplest form, all context search expressions are like this − a string of characters surrounded by slashes. Context searches are interchangeable with line numbers, so they can be used by themselves to find and print a desired line, or as line numbers for some other command, like s. They were used both ways in the examples above.

Suppose the buffer contains the three familiar lines Now is the time for all good men to come to the aid of their party. Then the *ed* line numbers /Now/+1 /good/ /party/−1 are all context search expressions, and they all refer to the same line (line 2). To make a change in line 2, you could say /Now/+1s/good/bad/ or /good/s/good/bad/ or /party/−1s/good/bad/ The choice is dictated only by convenience. You could print all three lines by, for instance /Now/,/party/p or /Now/,/Now/+2p or by any number of similar combinations. The first one of these might be better if you don't know how many lines are involved. (Of course, if there were only three lines in the buffer, you'd use 1,$p but not if there were several hundred.)

The basic rule is: a context search expression is *the same as* a line number, so it can be used wherever a line number is needed.

Exercise 6:

Experiment with context searching. Try a body of text with several occurrences of the same string of characters, and scan through it using the same context search.

Try using context searches as line numbers for the substitute, print and delete commands. (They can also be used with r, w, and a.)

Try context searching using ?text? instead of /text/. This scans lines in the buffer in reverse order rather than normal. This is sometimes useful if you go too far while looking for some string of characters − it's an easy way to back up.

(If you get funny results with any of the characters ˆ $ [* \ & read the section on ''Special Characters''.)

Ed provides a shorthand for repeating a context search for the same string. For example, the *ed* line number /string/ will find the next occurrence of string. It often happens that this is not the desired line, so the search must be repeated. This can be done by typing merely // This shorthand stands for ''the most recently used context search expression.'' It can also be used as the first string of the substitute command, as in /string1/s//string2/ which will find the next occurrence of string1 and replace it by string2. This can save a lot of typing. Similarly ?? means ''scan backwards for the same expression.''

Change and Insert − ''c'' and ''i''

This section discusses the *change* command c which is used to change or replace a group of one or more lines, and the *insert* command i which is used for inserting a group of one or more lines.

''Change'', written as c is used to replace a number of lines with different lines, which are typed in at the terminal. For example, to change lines through $ to something else, type .+1,$c . . . *type the lines of text you want here* . . .

The lines you type between the c command and the will take the place of the original lines between start line and end line. This is most useful in replacing a line or several lines which have errors in them.

If only one line is specified in the c command, then just that line is replaced. (You can type in as many replacement lines as you like.) Notice the use of to end the input − this works just like the in the append command and must appear by itself on a new line. If no line number is given, line dot is replaced. The value of dot is set to the last line you typed in.

''Insert'' is similar to append − for instance /string/i . . . *type the lines to be inserted here* . . .

will insert the given text *before* the next line that contains ''string''. The text between i and is *inserted before* the specified line. If no line number is specified dot is used. Dot is set to the last line inserted.

Exercise 7:

''Change'' is rather like a combination of delete followed by insert. Experiment to verify that *start, end* d i . . . *text* . . .

is almost the same as *start, end* c . . . *text* . . .

These are not *precisely* the same if line $ gets deleted. Check this out. What is dot?

Experiment with a and i, to see that they are similar, but not the same. You will observe that *line-number* a . . . *text* . . .

appends *after* the given line, while *line-number* i . . . *text* . . .

inserts *before* it. Observe that if no line number is given, i inserts before line dot, while a appends after line dot.

Moving text around: the ''m'' command

The move command m is used for cutting and pasting − it lets you move a group of lines from one place to another in the buffer. Suppose you want to put the first three lines of the buffer at the end instead. You could do it by saying: 1,3w temp $r temp 1,3d (Do you see why?) but you can do it a lot easier with the m command: 1,3m$ The general case is *start line, end line* m *after this line* Notice that there is a third line to be specified − the place where the moved stuff gets put. Of course the lines to be moved can be specified by context searches; if you had First paragraph . . . end of first paragraph. Second paragraph . . . end of second paragraph. you could reverse the two paragraphs like this: /Second/,/end of second/m/First/−1 Notice the −1: the moved text goes *after* the line mentioned. Dot gets set to the last line moved.

The global commands ‘‘g’’ and ‘‘v’’

The *global* command g is used to execute one or more *ed* commands on all those lines in the buffer that match some specified string. For example g/peling/p prints all lines that contain peling. More usefully, g/peling/s//pelling/gp makes the substitution everywhere on the line, then prints each corrected line. Compare this to 1,$s/peling/pelling/gp which only prints the last line substituted. Another subtle difference is that the g command does not give a ? if peling is not found where the s command will.

There may be several commands (including a, c, i, r, w, but not g); in that case, every line except the last must end with a backslash \: g/xxx/-1s/abc/def/\ +2s/ghi/jkl/\ -2,p makes changes in the lines before and after each line that contains xxx, then prints all three lines.

The v command is the same as g, except that the commands are executed on every line that does *not* match the string following v: v/ /d deletes every line that does not contain a blank.

Special Characters

You may have noticed that things just don’t work right when you used some characters like , *, $, and others in context searches and the substitute command. The reason is rather complex, although the cure is simple. Basically, *ed* treats these characters as special, with special meanings. For instance, *in a context search or the first string of the substitute command only,*

means ‘‘any character,’’ not a period, so /xy/ means ‘‘a line with an x, *any character,* and a y,’’ *not* just ‘‘a line with an x, a period, and a y.’’ A complete list of the special characters that can cause trouble is the following: ˆ $ [* \ *Warning:* The backslash character \ is special to *ed.* For safety’s sake, avoid it where possible. If you have to use one of the special characters in a substitute command, you can turn off its magic meaning temporarily by preceding it with the backslash. Thus s/*/backslash dot star/ will change \.* into ‘‘backslash dot star’’.

Here is a hurried synopsis of the other special characters. First, the circumflex ˆ signifies the beginning of a line. Thus /ˆstring/ finds string only if it is at the beginning of a line: it will find string but not the string... The dollar-sign $ is just the opposite of the circumflex; it means the end of a line: /string$/ will only find an occurrence of string that is at the end of some