Edit: A Tutorial

Ricki Blau

James Joyce

Computing Services
University of California
Berkeley, California 94720

ABSTRACT

This narrative introduction to the use of the text editor *edit* assumes no prior familiarity with computers or with text editing. Its aim is to lead the beginning UNIX† user through the fundamental steps of writing and revising a file of text. Edit, a version of the text editor *ex,* was designed to provide an informative environment for new and casual users.

We welcome comments and suggestions about this tutorial and the UNIX documentation in general.

September 1981

†UNIX is a trademark of Bell Laboratories.

Contents

Introduction

Text editing using a terminal connected to a computer allows you to create, modify, and print text easily. A *text editor* is a program that assists you as you create and modify text. The text editor you will learn here is named *edit.* Creating text using edit is as easy as typing it on an electric typewriter. Modifying text involves telling the text editor what you want to add, change, or delete. You can review your text by typing a command to print the file contents as they are currently. Another program (which we do not discuss in this document), a text formatter, rearranges your text for you into ‘‘finished form.’’

These lessons assume no prior familiarity with computers or with text editing. They consist of a series of text editing sessions which lead you through the fundamental steps of creating and revising text. After scanning each lesson and before beginning the next, you should try the examples at a terminal to get a feeling for the actual process of text editing. If you set aside some time for experimentation, you will soon become familiar with using the computer to write and modify text. In addition to the actual use of the text editor, other features of UNIX will be very important to your work. You can begin to learn about these other features by reading one of the other tutorials that provide a general introduction to the system. You will be ready to proceed with this lesson as soon as you are familiar with (1) your terminal and its special keys, (2) how to login, (3) and the ways of correcting typing errors. Let’s first define some terms:

program
: A set of instructions, given to the computer, describing the sequence of steps the computer performs in order to accomplish a specific task. The task must be specific, such as balancing your checkbook or editing your text. A general task, such as working for world peace, is something we can all do, but not something we can currently write programs to do.

UNIX
: UNIX is a special type of program, called an operating system, that supervises the machinery and all other programs comprising the total computer system.

edit
: *edit* is the name of the UNIX text editor you will be learning to use, and is a program that aids you in writing or revising text. Edit was designed for beginning users, and is a simplified version of an editor named *ex.*

file
: Each UNIX account is allotted space for the permanent storage of information, such as programs, data or text. A file is a logical unit of data, for example, an essay, a program, or a chapter from a book, which is stored on a computer system. Once you create a file, it is kept until you instruct the system to remove it. You may create a file during one UNIX session, end the session, and return to use it at a later time. Files contain anything you choose to write and store in them. The sizes of files vary to suit your needs; one file might hold only a single number, yet another might contain a very long document or program. The only way to save information from one session to the next is to store it in a file, which you will learn in Session 1.

filename
: Filenames are used to distinguish one file from another, serving the same purpose as the labels of manila folders in a file cabinet. In order to write or access information in a file, you use the name of that file in a UNIX command, and the system will automatically locate the file.

disk
: Files are stored on an input/output device called a disk, which looks something like a stack of phonograph records. Each surface is coated with a material similar to that on magnetic recording tape, and information is recorded on it.

buffer
: A temporary work space, made available to the user for the duration of a session of text editing and used for creating and modifying the text file. We can think of the buffer as a blackboard that is erased after each class, where each session with the editor is a class.

Session 1

Making contact with UNIX

To use the editor you must first make contact with the computer by logging in to UNIX. We'll quickly review the standard UNIX login procedure for the two ways you can make contact: on a terminal that is directly linked to the computer, or over a telephone line where the computer answers your call.

Directly-linked terminals

Turn on your terminal and press the RETURN key. You are now ready to login.

Dial-up terminals

If your terminal connects with the computer over a telephone line, turn on the terminal, dial the system access number, and, when you hear a high-pitched tone, place the telephone handset in the acoustic coupler, if you are using one. You are now ready to login.

Logging in

The message inviting you to login is:

login:

Type your login name, which identifies you to UNIX, on the same line as the login message, and press RETURN. If the terminal you are using has both upper and lower case, **be sure you enter your login name in lower case;** otherwise UNIX assumes your terminal has only upper case and will not recognize lower case letters you may type. UNIX types ‘‘login:’’ and you reply with your login name, for example ‘‘susan’’:

login: **susan** *(and press the RETURN key)*

(In the examples, input you would type appears in **bold face** to distinguish it from the responses from UNIX.)

UNIX will next respond with a request for a password as an additional precaution to prevent unauthorized people from using your account. The password will not appear when you type it, to prevent others from seeing it. The message is:

Password: *(type your password and press RETURN)*

If any of the information you gave during the login sequence was mistyped or incorrect, UNIX will respond with

Login incorrect.
login:

in which case you should start the login process anew. Assuming that you have successfully logged in, UNIX will print the message of the day and eventually will present you with a % at the beginning of a fresh line. The % is the UNIX prompt symbol which tells you that UNIX is ready to accept a command.

Asking for *edit*

You are ready to tell UNIX that you want to work with edit, the text editor. Now is a convenient time to choose a name for the file of text you are about to create. To begin your editing session, type **edit** followed by a space and then the filename you have selected; for example, ‘‘text’’. After that, press the RETURN key and wait for edit's response:

% **edit text** *(followed by a RETURN)*
"text" No such file or directory
:

If you typed the command correctly, you will now be in communication with edit. Edit has set aside a buffer for use as a temporary working space during your current editing session. Since ‘‘text’’ is a new file we are about to create the editor was unable to find that file, which it confirms by saying:

"text" No such file or directory

On the next line appears edit's prompt ‘‘:’’, announcing that you are in *command mode* and edit expects a command from you. You may now begin to create the new file.

The ‘‘Command not found’’ message

If you misspelled edit by typing, say, ‘‘editor’’, this might appear:

% **editor**
editor: Command not found
%

Your mistake in calling edit ‘‘editor’’ was treated by UNIX as a request for a program named ‘‘editor’’. Since there is no program named ‘‘editor’’, UNIX reported that the program was ‘‘not found’’. A new % indicates that UNIX is ready for another command, and you may then enter the correct command.

A summary

Your exchange with UNIX as you logged in and made contact with edit should look something like this:

login: **susan**
Password:
... A Message of General Interest ...
% **edit text**
"text" No such file or directory
:

Entering text

You may now begin entering text into the buffer. This is done by *appending* (or adding) text to whatever is currently in the buffer. Since there is nothing in the buffer at the moment, you are appending text to nothing; in effect, since you are adding text to nothing you are creating text. Most edit commands have two equivalent forms: a word that suggests what the command does, and a shorter abbreviation of that word. Many beginners find the full command names easier to remember at first, but once you are familiar with editing you may prefer to type the shorter abbreviations. The command to input text is ‘‘append’’. (It may be abbreviated ‘‘a’’.) Type **append** and press the RETURN key.

% **edit text**
: **append**

Messages from *edit*

If you make a mistake in entering a command and type something that edit does not recognize, edit will respond with a message intended to help you diagnose your error. For example, if you misspell the command to input text by typing, perhaps, ‘‘add’’ instead of ‘‘append’’ or ‘‘a’’, you will receive this message:

: **add**
add: Not an editor command
:

When you receive a diagnostic message, check what you typed in order to determine what part of your command confused edit. The message above means that edit was unable to recognize your mistyped command and, therefore, did not execute it. Instead, a new ‘‘:’’ appeared to let you know that edit is again ready to execute a command.

Text input mode

By giving the command ‘‘append’’ (or using the abbreviation ‘‘a’’), you entered *text input mode,* also known as *append mode.* When you enter text input mode, edit stops sending you a prompt. You will not receive any prompts or error messages while in text input mode. You can enter pretty much anything you want on the lines.

The lines are transmitted one by one to the buffer and held there during the editing session. You may append as much text as you want, and *when you wish to stop entering text lines you should type a period as the only character on the line and press the RETURN key*. When you type the period and press RETURN, you signal that you want to stop appending text, and edit responds by allowing you to exit text input mode and reenter command mode. Edit will again prompt you for a command by printing ‘‘:’’.

Leaving append mode does not destroy the text in the buffer. You have to leave append mode to do any of the other kinds of editing, such as changing, adding, or printing text. If you type a period as the first character and type any other character on the same line, edit will believe you want to remain in append mode and will not let you out. As this can be very frustrating, be sure to type **only** the period and the RETURN key.

This is a good place to learn an important lesson about computers and text: a blank space is a character as far as a computer is concerned. If you so much as type a period followed by a blank (that is, type a period and then the space bar on the keyboard), you will remain in append mode with the last line of text being:

.

Let’s say that you enter the lines (try to type **exactly** what you see, including ‘‘thiss’’):

This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
.

The last line is the period followed by a RETURN that gets you out of append mode.

Making corrections

If you have read a general introduction to UNIX, you will recall that it is possible to erase individual letters that you have typed. This is done by typing the designated erase character as many times as there are characters you want to erase.

The usual erase character varies from place to place and user to user. Often it is the backspace (control-H), so you can correct typing errors in the line you are typing by holding down the CTRL key and typing the ‘‘H’’ key. (Sometimes it is the DEL key.) If you type the erase character you will notice that the terminal backspaces in the line you are on. You can backspace over your error, and then type what you want to be the rest of the line.

If you make a bad start in a line and would like to begin again, you can either backspace to the beginning of the line or you can use the at-sign ‘‘@’’ to erase everything on the line:

Text edtiing is strange, but@
Text editing is strange, but nice.

When you type the at-sign (@), you erase the entire line typed so far and are given a fresh line to type on. You may immediately begin to retype the line. This, unfortunately, does not work after you type the line and press RETURN. To make corrections in lines that have been completed, it is necessary to use the editing commands covered in the next sessions.

Writing text to disk

You are now ready to edit the text. One common operation is to write the text to disk as a file for safekeeping after the session is over. This is the only way to save information from one session to the next, since the editor’s buffer is temporary and will last only until the end of the editing session. Learning how to write a file to disk is second in importance only to entering the text. To write the contents of the buffer to a disk file, use the command ‘‘write’’ (or its abbreviation ‘‘w’’):

:**write**

Edit will copy the contents of the buffer to a disk file. If the file does not yet exist, a new file will be created automatically and the presence of a ‘‘[New file]’’ will be noted. The newly-created file will be given the name specified when you entered the editor, in this case ‘‘text’’. To confirm that the disk file has been successfully written, edit will repeat the filename and give the number of lines and the total number of characters in the file. The buffer remains unchanged by the ‘‘write’’ command. All of the lines that were written to disk will still be in the buffer, should you want to modify or add to them.

Edit must have a name for the file to be written. If you forgot to indicate the name of the file when you began to edit, edit will print in response to your write command:

No current filename

If this happens, you can specify the filename in a new write command:

: **write text**

After the ‘‘write’’ (or ‘‘w’’), type a space and then the name of the file.

Signing off

We have done enough for this first lesson on using the UNIX text editor, and are ready to quit the session with edit. To do this we type ‘‘quit’’ (or ‘‘q’’) and press RETURN:

: **write**
"text" [New file] 3 lines, 90 characters
: **quit**
%

The % is from UNIX to tell you that your session with edit is over and you may command UNIX further. Since we want to end the entire session at the terminal, we also need to exit from UNIX. In response to the UNIX prompt of ‘‘ % ’’ type the command

% **logout**

This will end your session with UNIX, and will ready the terminal for the next user. It is always important to type **logout** at the end of a session to make absolutely sure no one could accidentally stumble into your abandoned session and thus gain access to your files, tempting even the most honest of souls.

This is the end of the first session on UNIX text editing.

Session 2

Login with UNIX as in the first session:

login: **susan** *(carriage return)*
Password: *(give password and carriage return)*
... A Message of General Interest ...
%

When you indicate you want to edit, you can specify the name of the file you worked on last time. This will start edit working, and it will fetch the contents of the file into the buffer, so that you can resume editing the same file. When edit has copied the file into the buffer, it will repeat its name and tell you the number of lines and characters it contains. Thus,

% edit text
"text" 3 lines, 90 characters
:

means you asked edit to fetch the file named ‘‘text’’ for editing, causing it to copy the 90 characters of text into the buffer. Edit awaits your further instructions, and indicates this by its prompt character, the colon (:). In this session, we will append more text to our file, print the contents of the buffer, and learn to change the text of a line.

Adding more text to the file

If you want to add more to the end of your text you may do so by using the append command to enter text input mode. When ‘‘append’’ is the first command of your editing session, the lines you enter are placed at the end of the buffer. Here we’ll use the abbreviation for the append command, ‘‘a’’:

: **a**
This is text added in Session 2.
It doesn’t mean much here, but
it does illustrate the editor.
.

You may recall that once you enter append mode using the ‘‘a’’ (or ‘‘append’’) command, you need to type a line containing only a period (.) to exit append mode.

Interrupt

Should you press the RUB key (sometimes labelled DELETE) while working with edit, it will send this message to you:

Interrupt
:

Any command that edit might be executing is terminated by rub or delete, causing edit to prompt you for a new command. If you are appending text at the time, you will exit from append mode and be expected to give another command. The line of text you were typing when the append command was interrupted will not be entered into the buffer.

Making corrections

If while typing the line you hit an incorrect key, recall that you may delete the incorrect character or cancel the entire line of input by erasing in the usual way. Refer either to the last few pages of Session 1 if you need to review the procedures for making a correction. The most important idea to remember is that erasing a character or cancelling a line must be done before you press the RETURN key.

Listing what's in the buffer (p)

Having appended text to what you wrote in Session 1, you might want to see all the lines in the buffer. To print the contents of the buffer, type the command:

: **1,$p**

The ‘‘1’’† stands for line 1 of the buffer, the ‘‘$’’ is a special symbol designating the last line of the buffer, and ‘‘p’’ (or **print**) is the command to print from line 1 to the end of the buffer. The command ‘‘1,$p’’ gives you:

This is some sample text.
And thiss is some more text.
Text editing is strange, but nice.
This is text added in Session 2.
It doesn't mean much here, but
it does illustrate the editor.

Occasionally, you may accidentally type a character that can't be printed, which can be done by striking a key while the CTRL key is pressed. In printing lines, edit uses a special notation to show the existence of non-printing characters. Suppose you had introduced the non-printing character ‘‘control-A’’ into the word ‘‘illustrate’’ by accidently pressing the CTRL key while typing ‘‘a’’. This can happen on many terminals because the CTRL key and the ‘‘A’’ key are beside each other. If your finger presses between the two keys, control-A results. When asked to print the contents of the buffer, edit would display

it does illustr^Ate the editor.

To represent the control-A, edit shows ‘‘^A’’. The sequence ‘‘^’’ followed by a capital letter stands for the one character entered by holding down the CTRL key and typing the letter which appears after the ‘‘^’’. We'll soon discuss the commands that can be used to correct this typing error.

In looking over the text we see that ‘‘this’’ is typed as ‘‘thiss’’ in the second line, a deliberate error so we can learn to make corrections. Let's correct the spelling.

Finding things in the buffer

In order to change something in the buffer we first need to find it. We can find ‘‘thiss’’ in the text we have entered by looking at a listing of the lines. Physically speaking, we search the lines of text looking for ‘‘thiss’’ and stop searching when we have found it. The way to tell edit to search for something is to type it inside slash marks:

: **/thiss/**

By typing **/thiss/** and pressing RETURN, you instruct edit to search for ‘‘thiss’’. If you ask edit to look for a pattern of characters which it cannot find in the buffer, it will respond ‘‘Pattern not found’’. When edit finds the characters ‘‘thiss’’, it will print the line of text for your inspection:

And thiss is some more text.

Edit is now positioned in the buffer at the line it just printed, ready to make a change in the line.

†The numeral ‘‘one’’ is the top left-most key, and should not be confused with the letter ‘‘el’’.

The current line

Edit keeps track of the line in the buffer where it is located at all times during an editing session. In general, the line that has been most recently printed, entered, or changed is the current location in the buffer. The editor is prepared to make changes at the current location in the buffer, unless you direct it to another location.

In particular, when you bring a file into the buffer, you will be located at the last line in the file, where the editor left off copying the lines from the file to the buffer. If your first editing command is ‘‘append’’, the lines you enter are added to the end of the file, after the current line — the last line in the file.

You can refer to your current location in the buffer by the symbol period (.) usually known by the name ‘‘dot’’. If you type ‘‘.’’ and carriage return you will be instructing edit to print the current line:

```
:.
And thiss is some more text.
```

If you want to know the number of the current line, you can type **.=** and press RETURN, and edit will respond with the line number:

```
:.=
2
```

If you type the number of any line and press RETURN, edit will position you at that line and print its contents:

```
:2
And thiss is some more text.
```

You should experiment with these commands to gain experience in using them to make changes.

Numbering lines (nu)

The **number (nu)** command is similar to print, giving both the number and the text of each printed line. To see the number and the text of the current line type

```
:nu
     2  And thiss is some more text.
```

Note that the shortest abbreviation for the number command is ‘‘nu’’ (and not ‘‘n’’, which is used for a different command). You may specify a range of lines to be listed by the number command in the same way that lines are specified for print. For example, **1,$nu** lists all lines in the buffer with their corresponding line numbers.

Substitute command (s)

Now that you have found the misspelled word, you can change it from ‘‘thiss’’ to ‘‘this’’. As far as edit is concerned, changing things is a matter of substituting one thing for another. As *a* stood for *append,* so *s* stands for *substitute.* We will use the abbreviation ‘‘s’’ to reduce the chance of mistyping the substitute command. This command will instruct edit to make the change:

2s/thiss/this/

We first indicate the line to be changed, line 2, and then type an ‘‘s’’ to indicate we want edit to make a substitution. Inside the first set of slashes are the characters that we want to change, followed by the characters to replace them, and then a closing slash mark. To summarize:

2s/ *what is to be changed* / *what to change it to* /

If edit finds an exact match of the characters to be changed it will make the change **only** in the first occurrence of the characters. If it does not find the characters to be changed, it will respond:

Substitute pattern match failed

indicating that your instructions could not be carried out. When edit does find the characters that you want to change, it will make the substitution and automatically print the changed line, so that you can check that the correct substitution was made. In the example,

: **2s/thiss/this/**
And this is some more text.

line 2 (and line 2 only) will be searched for the characters ‘‘thiss’’, and when the first exact match is found, ‘‘thiss’’ will be changed to ‘‘this’’. Strictly speaking, it was not necessary above to specify the number of the line to be changed. In

: **s/thiss/this/**

edit will assume that we mean to change the line where we are currently located (‘‘.’’). In this case, the command without a line number would have produced the same result because we were already located at the line we wished to change.

For another illustration of the substitute command, let us choose the line:

Text editing is strange, but nice.

You can make this line a bit more positive by taking out the characters ‘‘strange, but ’’ so the line reads:

Text editing is nice.

A command that will first position edit at the desired line and then make the substitution is:

: **/strange/s/strange, but //**

What we have done here is combine our search with our substitution. Such combinations are perfectly legal, and speed up editing quite a bit once you get used to them. That is, you do not necessarily have to use line numbers to identify a line to edit. Instead, you may identify the line you want to change by asking edit to search for a specified pattern of letters that occurs in that line. The parts of the above command are:

/strange/	tells edit to find the characters ‘‘strange’’ in the text
s	tells edit to make a substitution
/strange, but //	substitutes nothing at all for the characters ‘‘strange, but ’’

You should note the space after ‘‘but’’ in ‘‘/strange, but /’’. If you do not indicate that the space is to be taken out, your line will read:

Text editing is nice.

which looks a little funny because of the extra space between ‘‘is’’ and ‘‘nice’’. Again, we realize from this that a blank space is a real character to a computer, and in editing text we need to be aware of spaces within a line just as we would be aware of an ‘‘a’’ or a ‘‘4’’.

Another way to list what’s in the buffer (z)

Although the print command is useful for looking at specific lines in the buffer, other commands may be more convenient for viewing large sections of text. You can ask to see a screen full of text at a time by using the command **z.** If you type

: **1z**

edit will start with line 1 and continue printing lines, stopping either when the screen of your terminal is full or when the last line in the buffer has been printed. If you want to read the next segment of text, type the command

: **z**

If no starting line number is given for the z command, printing will start at the ‘‘current’’ line, in this case the last line printed. Viewing lines in the buffer one screen full at a time is known as *paging*. Paging can also be used to print a section of text on a hard-copy terminal.

Saving the modified text

This seems to be a good place to pause in our work, and so we should end the second session. If you (in haste) type ‘‘q’’ to quit the session your dialogue with edit will be:

: **q**
No write since last change (:quit! overrides)
:

This is edit’s warning that you have not written the modified contents of the buffer to disk. You run the risk of losing the work you did during the editing session since you typed the latest write command. Because in this lesson we have not written to disk at all, everything we have done would have been lost if edit had obeyed the **q** command. If you did not want to save the work done during this editing session, you would have to type ‘‘q!’’ or (‘‘quit!’’) to confirm that you indeed wanted to end the session immediately, leaving the file as it was after the most recent ‘‘write’’ command. However, since you want to save what you have edited, you need to type:

: **w**
"text" 6 lines, 171 characters

and then follow with the commands to quit and logout:

: **q**
% logout

and hang up the phone or turn off the terminal when UNIX asks for a name. Terminals connected to the port selector will stop after the logout command, and pressing keys on the keyboard will do nothing.

This is the end of the second session on UNIX text editing.

Session 3

Bringing text into the buffer (e)

Login to UNIX and make contact with edit. You should try to login without looking at the notes, but if you must then by all means do.

Did you remember to give the name of the file you wanted to edit? That is, did you type

% **edit text**

or simply

% **edit**

Both ways get you in contact with edit, but the first way will bring a copy of the file named ‘‘text’’ into the buffer. If you did forget to tell edit the name of your file, you can get it into the buffer by typing:

: **e text**
"text" 6 lines, 171 characters

The command **edit,** which may be abbreviated **e**, tells edit that you want to erase anything that might already be in the buffer and bring a copy of the file ‘‘text’’ into the buffer for editing. You may also use the edit (e) command to change files in the middle of an editing session, or to give edit the name of a new file that you want to create. Because the edit command clears the buffer, you will receive a warning if you try to edit a new file without having saved a copy of the old file. This gives you a chance to write the contents of the buffer to disk before editing the next file.

Moving text in the buffer (m)

Edit allows you to move lines of text from one location in the buffer to another by means of the **move** (**m**) command. The first two examples are for illustration only, though after you have read this Session you are welcome to return to them for practice. The command

: **2,4m$**

directs edit to move lines 2, 3, and 4 to the end of the buffer ($). The format for the move command is that you specify the first line to be moved, the last line to be moved, the move command ‘‘m’’, and the line after which the moved text is to be placed. So,

: **1,3m6**

would instruct edit to move lines 1 through 3 (inclusive) to a location after line 6 in the buffer. To move only one line, say, line 4, to a location in the buffer after line 5, the command would be ‘‘4m5’’.

Let’s move some text using the command:

: **5,$m1**
2 lines moved
it does illustrate the editor.

After executing a command that moves more than one line of the buffer, edit tells how many lines were affected by the move and prints the last moved line for your inspection. If you want to see more than just the last line, you can then use the print (p), z, or number (nu) command to view more text. The buffer should now contain:

This is some sample text.
It doesn’t mean much here, but
it does illustrate the editor.
And this is some more text.
Text editing is nice.
This is text added in Session 2.

You can restore the original order by typing:

: **4,$m1**

or, combining context searching and the move command:

: **/And this is some/,/This is text/m/This is some sample/**

(Do not type both examples here!) The problem with combining context searching with the move command is that your chance of making a typing error in such a long command is greater than if you type line numbers.

Copying lines (copy)

The **copy** command is used to make a second copy of specified lines, leaving the original lines where they were. Copy has the same format as the move command, for example:

: **2,5copy $**

makes a copy of lines 2 through 5, placing the added lines after the buffer's end ($). Experiment with the copy command so that you can become familiar with how it works. Note that the shortest abbreviation for copy is **co** (and not the letter ‘‘c’’, which has another meaning).

Deleting lines (d)

Suppose you want to delete the line

This is text added in Session 2.

from the buffer. If you know the number of the line to be deleted, you can type that number followed by **delete** or **d**. This example deletes line 4, which is ‘‘This is text added in Session 2.’’ if you typed the commands suggested so far.

: **4d**
It doesn't mean much here, but

Here ‘‘4’’ is the number of the line to be deleted, and ‘‘delete’’ or ‘‘d’’ is the command to delete the line. After executing the delete command, edit prints the line that has become the current line (‘‘.’’).

If you do not happen to know the line number you can search for the line and then delete it using this sequence of commands:

: **/added in Session 2./**
This is text added in Session 2.
: **d**
It doesn't mean much here, but

The ‘‘/added in Session 2./’’ asks edit to locate and print the line containing the indicated text, starting its search at the current line and moving line by line until it finds the text. Once you are sure that you have correctly specified the line you want to delete, you can enter the delete (d) command. In this case it is not necessary to specify a line number before the ‘‘d’’. If no line number is given, edit deletes the current line (‘‘.’’), that is, the line found by our search. After the deletion, your buffer should contain:

This is some sample text.
And this is some more text.
Text editing is nice.
It doesn't mean much here, but
it does illustrate the editor.
And this is some more text.
Text editing is nice.
This is text added in Session 2.
It doesn't mean much here, but

To delete both lines 2 and 3:

And this is some more text.
Text editing is nice.

you type

: **2,3d**
2 lines deleted

which specifies the range of lines from 2 to 3, and the operation on those lines — ‘‘d’’ for delete. If you delete more than one line you will receive a message telling you the number of lines deleted, as indicated in the example above.

The previous example assumes that you know the line numbers for the lines to be deleted. If you do not you might combine the search command with the delete command:

: **/And this is some/,/Text editing is nice./d**

A word or two of caution

In using the search function to locate lines to be deleted you should be **absolutely sure** the characters you give as the basis for the search will take edit to the line you want deleted. Edit will search for the first occurrence of the characters starting from where you last edited – that is, from the line you see printed if you type dot (.).

A search based on too few characters may result in the wrong lines being deleted, which edit will do as easily as if you had meant it. For this reason, it is usually safer to specify the search and then delete in two separate steps, at least until you become familiar enough with using the editor that you understand how best to specify searches. For a beginner it is not a bad idea to double-check each command before pressing RETURN to send the command on its way.

Undo (u) to the rescue

The **undo (u)** command has the ability to reverse the effects of the last command that changed the buffer. To undo the previous command, type ‘‘u’’ or ‘‘undo’’. Undo can rescue the contents of the buffer from many an unfortunate mistake. However, its powers are not unlimited, so it is still wise to be reasonably careful about the commands you give.

It is possible to undo only commands which have the power to change the buffer — for example, delete, append, move, copy, substitute, and even undo itself. The commands write (w) and edit (e), which interact with disk files, cannot be undone, nor can commands that do not change the buffer, such as print. Most importantly, the **only** command that can be reversed by undo is the last ‘‘undo-able’’ command you typed. You can use control-H and @ to change commands while you are typing them, and undo to reverse the effect of the commands after you have typed them and pressed RETURN.

To illustrate, let’s issue an undo command. Recall that the last buffer-changing command we gave deleted the lines formerly numbered 2 and 3. Typing undo at this moment will reverse the effects of the deletion, causing those two lines to be replaced in the buffer.

: **u**
2 more lines in file after undo
And this is some more text.

Here again, edit informs you if the command affects more than one line, and prints the text of the line which is now ‘‘dot’’ (the current line).

More about the dot (.) and buffer end ($)

The function assumed by the symbol dot depends on its context. It can be used:

1. to exit from append mode; we type dot (and only a dot) on a line and press RETURN;
2. to refer to the line we are at in the buffer.

Dot can also be combined with the equal sign to get the number of the line currently being edited:

: **.=**

If we type ‘‘**.**=’’ we are asking for the number of the line, and if we type ‘‘**.**’’ we are asking for the text of the line.

In this editing session and the last, we used the dollar sign to indicate the end of the buffer in commands such as print, copy, and move. The dollar sign as a command asks edit to print the last line in the buffer. If the dollar sign is combined with the equal sign (**$=**) edit will print the line number corresponding to the last line in the buffer.

‘‘**.**’’ and ‘‘$’’, then, represent line numbers. Whenever appropriate, these symbols can be used in place of line numbers in commands. For example

: **.,$d**

instructs edit to delete all lines from the current line (**.**) to the end of the buffer.

Moving around in the buffer (+ and −)

When you are editing you often want to go back and re-read a previous line. You could specify a context search for a line you want to read if you remember some of its text, but if you simply want to see what was written a few, say 3, lines ago, you can type

−3p

This tells edit to move back to a position 3 lines before the current line (.) and print that line. You can move forward in the buffer similarly:

+2p

instructs edit to print the line that is 2 ahead of your current position.

You may use ‘‘+’’ and ‘‘−’’ in any command where edit accepts line numbers. Line numbers specified with ‘‘+’’ or ‘‘−’’ can be combined to print a range of lines. The command

: **−1,+2copy$**

makes a copy of 4 lines: the current line, the line before it, and the two after it. The copied lines will be placed after the last line in the buffer ($), and the original lines referred to by ‘‘−1’’ and ‘‘+2’’ remain where they are.

Try typing only ‘‘−’’; you will move back one line just as if you had typed ‘‘−1p’’. Typing the command ‘‘+’’ works similarly. You might also try typing a few plus or minus signs in a row (such as ‘‘+++’’) to see edit’s response. Typing RETURN alone on a line is the equivalent of typing ‘‘+1p’’; it will move you one line ahead in the buffer and print that line.

If you are at the last line of the buffer and try to move further ahead, perhaps by typing a ‘‘+’’ or a carriage return alone on the line, edit will remind you that you are at the end of the buffer:

At end-of-file

or

Not that many lines in buffer

Similarly, if you try to move to a position before the first line, edit will print one of these messages:

Nonzero address required on this command

or

Negative address − first buffer line is 1

The number associated with a buffer line is the line’s ‘‘address’’, in that it can be used to locate the line.

Changing lines (c)

You can also delete certain lines and insert new text in their place. This can be accomplished easily with the **change (c)** command. The change command instructs edit to delete specified lines and then switch to text input mode to accept the text that will replace them. Let’s say you want to change the first two lines in the buffer:

```
This is some sample text.
And this is some more text.
```

to read

```
This text was created with the UNIX text editor.
```

To do so, you type:

:**1,2c**
2 lines changed
This text was created with the UNIX text editor.
.
:

In the command **1,2c** we specify that we want to change the range of lines beginning with 1 and ending with 2 by giving line numbers as with the print command. These lines will be deleted. After you type RETURN to end the change command, edit notifies you if more than one line will be changed and places you in text input mode. Any text typed on the following lines will be inserted into the position where lines were deleted by the change command. **You will remain in text input mode until you exit in the usual way, by typing a period alone on a line.** Note that the number of lines added to the buffer need not be the same as the number of lines deleted.

This is the end of the third session on text editing with UNIX.

Session 4

This lesson covers several topics, starting with commands that apply throughout the buffer, characters with special meanings, and how to issue UNIX commands while in the editor. The next topics deal with files: more on reading and writing, and methods of recovering files lost in a crash. The final section suggests sources of further information.

Making commands global (g)

One disadvantage to the commands we have used for searching or substituting is that if you have a number of instances of a word to change it appears that you have to type the command repeatedly, once for each time the change needs to be made. Edit, however, provides a way to make commands apply to the entire contents of the buffer – the **global (g)** command.

To print all lines containing a certain sequence of characters (say, ''text'') the command is:

: **g/text/p**

The ''g'' instructs edit to make a global search for all lines in the buffer containing the characters ''text''. The ''p'' prints the lines found.

To issue a global command, start by typing a ''g'' and then a search pattern identifying the lines to be affected. Then, on the same line, type the command to be executed for the identified lines. Global substitutions are frequently useful. For example, to change all instances of the word ''text'' to the word ''material'' the command would be a combination of the global search and the substitute command:

: **g/text/s/text/material/g**

Note the ''g'' at the end of the global command, which instructs edit to change each and every instance of ''text'' to ''material''. If you do not type the ''g'' at the end of the command only the *first* instance of ''text'' *in each line* will be changed (the normal result of the substitute command). The ''g'' at the end of the command is independent of the ''g'' at the beginning. You may give a command such as:

: **5s/text/material/g**

to change every instance of ''text'' in line 5 alone. Further, neither command will change ''text'' to ''material'' if ''Text'' begins with a capital rather than a lower-case *t.*

Edit does not automatically print the lines modified by a global command. If you want the lines to be printed, type a ''p'' at the end of the global command:

: **g/text/s/text/material/gp**

You should be careful about using the global command in combination with any other – in essence, be sure of what you are telling edit to do to the entire buffer. For example,

: **g/ /d**
72 less lines in file after global

will delete every line containing a blank anywhere in it. This could adversely affect your document, since most lines have spaces between words and thus would be deleted. After executing the global command, edit will print a warning if the command added or deleted more than one line. Fortunately, the undo command can reverse the effects of a global command. You should experiment with the global command on a small file of text to see what it can do for you.

More about searching and substituting

In using slashes to identify a character string that we want to search for or change, we have always specified the exact characters. There is a less tedious way to repeat the same string of characters. To change ''text'' to ''texts'' we may type either

: **/text/s/text/texts/**

as we have done in the past, or a somewhat abbreviated command:

: **/text/s//texts/**

In this example, the characters to be changed are not specified − there are no characters, not even a space, between the two slash marks that indicate what is to be changed. This lack of characters between the slashes is taken by the editor to mean ‘‘use the characters we last searched for as the characters to be changed.’’

Similarly, the last context search may be repeated by typing a pair of slashes with nothing between them:

: **/does/**
It doesn’t mean much here, but
: **//**
it does illustrate the editor.

(You should note that the search command found the characters ‘‘does’’ in the word ‘‘doesn’t’’ in the first search request.) Because no characters are specified for the second search, the editor scans the buffer for the next occurrence of the characters ‘‘does’’.

Edit normally searches forward through the buffer, wrapping around from the end of the buffer to the beginning, until the specified character string is found. If you want to search in the reverse direction, use question marks (?) instead of slashes to surround the characters you are searching for.

It is also possible to repeat the last substitution without having to retype the entire command. An ampersand (&) used as a command repeats the most recent substitute command, using the same search and replacement patterns. After altering the current line by typing

: **s/text/texts/**

you type

: **/text/&**

or simply

: **//&**

to make the same change on the next line in the buffer containing the characters ‘‘text’’.

Special characters

Two characters have special meanings when used in specifying searches: ‘‘$’’ and ‘‘^’’. ‘‘$’’ is taken by the editor to mean ‘‘end of the line’’ and is used to identify strings that occur at the end of a line.

: **g/text.$/s//material./p**

tells the editor to search for all lines ending in ‘‘text.’’ (and nothing else, not even a blank space), to change each final ‘‘text.’’ to ‘‘material.’’, and print the changed lines.

The symbol ‘‘^’’ indicates the beginning of a line. Thus,

: **s/^/1. /**

instructs the editor to insert ‘‘1.’’ and a space at the beginning of the current line.

The characters ‘‘$’’ and ‘‘^’’ have special meanings only in the context of searching. At other times, they are ordinary characters. If you ever need to search for a character that has a special meaning, you must indicate that the character is to lose temporarily its special significance by typing another special character, the backslash (\), before it.

: **s/\$/dollar/**

looks for the character ‘‘$’’ in the current line and replaces it by the word ‘‘dollar’’. Were it not for the backslash, the ‘‘$’’ would have represented ‘‘the end of the line’’ in your search rather than the character ‘‘$’’. The backslash retains its special significance unless it is preceded by another backslash.

Issuing UNIX commands from the editor

After creating several files with the editor, you may want to delete files no longer useful to you or ask for a list of your files. Removing and listing files are not functions of the editor, and so they require the use of UNIX system commands (also referred to as ‘‘shell’’ commands, as ‘‘shell’’ is the name of the program that processes UNIX commands). You do not need to quit the editor to execute a UNIX command as long as you indicate that it is to be sent to the shell for execution. To use the UNIX command **rm** to remove the file named ‘‘junk’’ type:

> : **!rm junk**
> !
> :

The exclamation mark (!) indicates that the rest of the line is to be processed as a shell command. If the buffer contents have not been written since the last change, a warning will be printed before the command is executed:

> [No write since last change]

The editor prints a ‘‘!’’ when the command is completed. Other tutorials describe useful features of the system, of which an editor is only one part.

Filenames and file manipulation

Throughout each editing session, edit keeps track of the name of the file being edited as the *current filename.* Edit remembers as the current filename the name given when you entered the editor. The current filename changes whenever the edit (e) command is used to specify a new file. Once edit has recorded a current filename, it inserts that name into any command where a filename has been omitted. If a write command does not specify a file, edit, as we have seen, supplies the current filename. If you are editing a file named ‘‘draft3’’ having 283 lines in it, you can have the editor write onto a different file by including its name in the write command:

> : **w chapter3**
> "chapter3" [new file] 283 lines, 8698 characters

The current filename remembered by the editor *will not be changed as a result of the write command.* Thus, if the next write command does not specify a name, edit will write onto the current file (‘‘draft3’’) and not onto the file ‘‘chapter3’’.

The file (f) command

To ask for the current filename, type **file** (or **f**). In response, the editor provides current information about the buffer, including the filename, your current position, the number of lines in the buffer, and the percent of the distance through the file your current location is.

> : **f**
> "text" [Modified] line 3 of 4 --75%--

If the contents of the buffer have changed since the last time the file was written, the editor will tell you that the file has been ‘‘[Modified]’’. After you save the changes by writing onto a disk file, the buffer will no longer be considered modified:

> : **w**
> "text" 4 lines, 88 characters
> : **f**
> "text" line 3 of 4 --75%--

Reading additional files (r)

The **read (r)** command allows you to add the contents of a file to the buffer at a specified location, essentially copying new lines between two existing lines. To use it, specify the line after which the new text will be placed, the **read (r)** command, and then the name of the file. If you have a file named ‘‘example’’, the command

: **$r example**
"example" 18 lines, 473 characters

reads the file ‘‘example’’ and adds it to the buffer after the last line. The current filename is not changed by the read command.

Writing parts of the buffer

The **write (w)** command can write all or part of the buffer to a file you specify. We are already familiar with writing the entire contents of the buffer to a disk file. To write only part of the buffer onto a file, indicate the beginning and ending lines before the write command, for example

: **45,$w ending**

Here all lines from 45 through the end of the buffer are written onto the file named *ending.* The lines remain in the buffer as part of the document you are editing, and you may continue to edit the entire buffer. Your original file is unaffected by your command to write part of the buffer to another file. Edit still remembers whether you have saved changes to the buffer in your original file or not.

Recovering files

Although it does not happen very often, there are times UNIX stops working because of some malfunction. This situation is known as a *crash*. Under most circumstances, edit’s crash recovery feature is able to save work to within a few lines of changes before a crash (or an accidental phone hang up). If you lose the contents of an editing buffer in a system crash, you will normally receive mail when you login that gives the name of the recovered file. To recover the file, enter the editor and type the command **recover** (**rec**), followed by the name of the lost file. For example, to recover the buffer for an edit session involving the file ‘‘chap6’’, the command is:

: **recover chap6**

Recover is sometimes unable to save the entire buffer successfully, so always check the contents of the saved buffer carefully before writing it back onto the original file. For best results, write the buffer to a new file temporarily so you can examine it without risk to the original file. Unfortunately, you cannot use the recover command to retrieve a file you removed using the shell command **rm**.

Other recovery techniques

If something goes wrong when you are using the editor, it may be possible to save your work by using the command **preserve** (**pre**), which saves the buffer as if the system had crashed. If you are writing a file and you get the message ‘‘Quota exceeded’’, you have tried to use more disk storage than is allotted to your account. *Proceed with caution* because it is likely that only a part of the editor’s buffer is now present in the file you tried to write. In this case you should use the shell escape from the editor (!) to remove some files you don’t need and try to write the file again. If this is not possible and you cannot find someone to help you, enter the command

: **preserve**

and wait for the reply,

File preserved.

If you do not receive this reply, seek help immediately. Do not simply leave the editor. If you do, the buffer will be lost, and you may not be able to save your file. If the reply is ‘‘File preserved.’’ you can leave the editor (or logout) to remedy the situation. After a preserve, you can use the recover command once the problem has been corrected, or the **−r** option of the edit command if you leave the editor and want to return.

If you make an undesirable change to the buffer and type a write command before discovering your mistake, the modified version will replace any previous version of the file. Should you ever lose a good version of a document in this way, do not panic and leave the editor. As long as you stay in the editor, the contents of the buffer remain accessible. Depending on the nature of the problem, it may be possible to restore the buffer to a more complete state with the undo command. After fixing the damaged buffer, you can again write the file to disk.

Further reading and other information

Edit is an editor designed for beginning and casual users. It is actually a version of a more powerful editor called *ex.* These lessons are intended to introduce you to the editor and its more commonly-used commands. We have not covered all of the editor's commands, but a selection of commands that should be sufficient to accomplish most of your editing tasks. You can find out more about the editor in the *Ex Reference Manual,* which is applicable to both *ex* and *edit.* One way to become familiar with the manual is to begin by reading the description of commands that you already know.

Using *ex*

As you become more experienced with using the editor, you may still find that edit continues to meet your needs. However, should you become interested in using *ex,* it is easy to switch. To begin an editing session with *ex,* use the name **ex** in your command instead of **edit.**

Edit commands also work in *ex,* but the editing environment is somewhat different. You should be aware of a few differences between *ex* and *edit.* In edit, only the characters ‘‘^’’, ‘‘$’’, and ‘‘\’’ have special meanings in searching the buffer or indicating characters to be changed by a substitute command. Several additional characters have special meanings in ex, as described in the *Ex Reference Manual.* Another feature of the edit environment prevents users from accidently entering two alternative modes of editing, *open* and *visual,* in which the editor behaves quite differently from normal command mode. If you are using ex and you encounter strange behavior, you may have accidently entered open mode by typing ‘‘o’’. Type the ESC key and then a ‘‘Q’’ to get out of open or visual mode and back into the regular editor command mode. The document *An Introduction to Display Editing with Vi* provide full details of visual mode.

Index