


Special: Strategies to Secure Your Mobile Workforce


 Search 

Vendor Offer

**Gotta Gadget** Check them out here.

Home Webcasts White Papers Newsletters News Topic map Careers

Changing the way you view IT

## Grep this

Unix Insider 11/29/99

*Using grep, fgrep, and egrep to search for strings of words*

The `grep` utility, which allows files to be searched for strings of words, uses a syntax similar to the regular expression syntax of the `vi`, `ex`, `ed`, and `sed` editors. `grep` comes in three flavors, `grep`, `fgrep`, and `egrep`, all of which I'll cover in this article.

The name `grep` is derived from the editor command `g/re/p`, which literally translates to "globally search for a regular wxpression and print what you find." Regular expressions are at the core of `grep`, and I'll cover them after a brief description of some of the utility's command options.

The simplest `grep` command is `grep (search pattern) (files list)`, as in:

```
grep hello *
```

The output of this command might be something like this:

```
$ grep hello *
story.txt: so I said hello and she smiled back
intro.txt: use the hello.c program as an example of C programming
$
```

`grep` is case sensitive, so in order to change the search to include "hello," "Hello," or "HELLO," use the `-y` or `-i` option. Earlier versions of `grep` used `-y`, and later versions use `-i`. `-y` is now considered obsolete, although some versions of `grep` do support both. In the following example, more hellos show up because the search is case independent.

```
$ grep -i hello *
story.txt: so I said hello and she smiled back
story.txt: I could hear my echo, "HELLO."
intro.txt: use the hello.c program as an example of C programming
hello.c:     printf("Hello, world. \n");
$
```

This command searches all files in the current directory and prints the file name and the line containing the string "hello" for any files that contain that string.

The output of `grep` varies depending on whether you're searching one or several files. If only one file is named on the command line, the output doesn't include the file name, as in the following example:

```
$ grep -i hello hello.c
    printf("Hello, world. \n");
$
```

The one-file rule applies whether you use a wild card in your file list or not. If `hello.c` were the only file in the current directory, using a wild card to locate the file would still produce an unnamed file output. In the following example, the user is searching for any C files containing "hello." There is only one C file in the directory, so the output is identical to the previous example.

```
$ grep -i hello *.c
    printf("Hello, world. \n");
$
```

I don't know of a `grep` that has a work-around for this behavior, but you could use the `-l` option instead,

which prints the file name only and not the line containing the string. At least you would know the name of the file that contained the string.

```
$ grep -il hello *.c
hello.c:
$
```

The `-l` option can be used to extract a list of files containing the string. The file name is printed only once, even though the string may appear in multiple lines within that file. In the following example, `story.txt` appears only once, even though it contains more than one "hello."

```
$ grep -il hello *
hello.c:
intro.txt:
story.txt:
$
```

The `-l` option suppresses most of the other output options from `grep`. On the other hand, the `-n` option will print a line number as well as the text, as in the following example:

```
$ grep -in hello *
hello.c:7:    printf("Hello, world. \n");
intro.txt:44: use the hello.c program as an example of C programming
story.txt:110: so I said hello and she smiled back
story.txt:187: I could hear my echo, "HELLO."
$
```

The `-v` option outputs the complement of the search, i.e., all lines *not* containing the requested search pattern.

```
$ grep -iv hello intro.txt
You will be able to get more practice if you
at its simplest
$
```

The `-c` option prints only a count of lines matched. It also has the interesting and useful side effect of listing all the files it searches, not just the successful hits.

```
$ grep -ic hello *
data.txt:0
hello.c:1
intro.txt:1
intro2.txt:0
story.txt:2
$
```

Some versions of `grep` come with `-r` as an option, which prompts `grep` to search recursively through subdirectories. The default behavior is to search only one directory, so the `-r` option, as provided in GNU and other implementations of `grep`, is the exception rather than the rule.

## Going wild with grep

So far I've covered some of the input and output options, but the real power of `grep` is in its search pattern, which uses regular expressions. `grep` can match simple strings, as we saw in the "hello" example we played with above; but it can also use a variety of wild cards and special symbols to create a regular expression to search for more complex strings.

I will begin with some of the simpler characters in a regular expression. A `^` (caret) character means the start of a line and a `$` (dollar) character means the end of one.

The wild cards used by `grep` frequently clash with the special symbols that the shell uses, so the usual practice is to enclose complex search strings within single quotes. The two following examples would match any case version of "hello" at the start and end of a line, respectively.

```
$ grep '^hello' *

$ grep 'hello$' *
```

The dot or period character (`.`) will match any single character. For example, the following would match

any character followed by "ello," as in "aello," "bello," "cello," and so on all the way through "zello." Odd combinations, like "1ello" and "?ello," would also be included; any combination of one initial character followed by "ello" is valid. The dot does not match the beginning or end of a line; therefore, "ello" at the start of a line would not be matched.

```
$ grep '.ello' *
```

Optional characters can be enclosed in square brackets (`[ ]`) causing any of the enclosed characters to be matched. The following search string would match "hello," "cello," or "jello."

```
$ grep '[hcj]ello' *
```

Optional characters can also be specified by using a range consisting of two characters separated by a hyphen. The following example would match "bay," "cay," or "day."

```
$ grep '[b-d]ay' *
```

An optional character or range of characters can be preceded by a caret (`^`) to invert the sense of the match. The following would match any character preceded by "ay" *except* the combinations "bay," "cay," and "day."

```
$ grep '[^b-d]ay' *
```

Note that options and ranges represent a match of a single character.

Any single character match (including a single character matched by a option/range specification) can be repeated by using the asterisk character (`*`). An asterisk following a single character means "zero or more occurrences" of the preceding match. The following search requests any line containing "hello" followed by "dolly" where the words are separated by zero or more spaces. Note that the asterisk follows the space after "hello" and therefore applies to the space character.

```
$ grep 'hello *dolly' *
```

This search would match any of the following, without regard to the number of spaces between the words.

```
hellodolly
hello dolly
hello          dolly
```

The asterisk can be applied to an option or range. Following search matches "c" and "t" with any number of vowels (or no vowels) in between.

```
$ grep 'c[aeiou]*t' somewords.txt
cat
coat
coot
cot
cout
cut
ct
$
```

## Extending grep

At this point `grep` and `egrep` depart from one another. `egrep` stands for extended `grep`. The POSIX 1003.2 standard defined a set of regular expression characters, called *modern*, *extended*, or *full* regular expressions. The regular expressions I cited earlier are frequently called *older* or *basic* regular expressions. There is some overlap between the two, and recent versions of `grep` can be made to behave like `egrep` by using the `-E` option.

The `egrep` utility uses extended regular expressions, with a useful one being the plus (`+`) character, which works like the asterisk (`*`) but means "one or more" rather than "zero or more." Using `egrep` in the above example with a `+` instead of an `*` would cause the search to exclude "ct" because it doesn't contain one or more vowels.

```
$ egrep 'c[aeiou]+t' somewords.txt
cat
```

```
coat
coot
cot
cout
cut
$
```

If you use `grep` to achieve the same results, the search pattern becomes clumsier. The next example asks for "c," followed by any vowel, followed by zero or more occurrences of any vowel, followed by "t."

```
$ grep 'c[aeiou][aeiou]*t' somewords.txt
cat
coat
coot
cot
cout
cut
$
```

The `egrep` utility also adds a question mark (?), meaning zero or one occurrence, as another version of multiple occurrence matching.

```
* = zero or more occurrences
+ = one or more occurrences
? = zero or one occurrence
```

The vertical bar (|) creates an "or" condition between two possible search patterns. In the following example, `egrep` searches for "c," followed by one or more vowels, followed by "t," or for "p" followed by one or more vowels, followed by "l." Because the search string doesn't specify that the word must end after the closing "t" or "l," this example has matched "paula" and "paella," as well as words that end in "l."

```
$ egrep 'c[aeiou]+t|p[aeiou]+l' somewords.txt
cat
coat
coot
cot
cut
cet
cit
pal
paella
paul
paula
peal
peel
pool
$
```

You can fudge this with `grep` by entering multiple search patterns and inserting newlines in between the patterns. This can be used with `egrep` and `fgrep` as well, but I'm introducing it here simply to highlight the difficulty of imitating `egrep` with `grep` when it would be simpler to use `egrep`.

In the following example, the first part of the command is entered on one line, and then Enter is pressed while the single quotes are still open. The shell prompts for additional input and continues to accept lines until the closing quote appears. Each individual line represents a separate search string to `grep`. This trick is useful with any version of `grep`.

```
$ grep 'c[aeiou][aeiou]*t
> p[aeiou][aeiou]*l' somewords.txt
cat
coat
coot
cot
cut
cet
cit
pal
paella
paul
paula
```

```
peal
peel
pool
$
```

With `egrep`, simple parentheses can be used to group sections of a search pattern together. In the following example, the search pattern will match any of the words shown in the result list. The parentheses group "[Ss]ome" and "[Aa]ny" are optional strings, followed by "one."

```
$ egrep '([Ss]ome|[Aa]ny)one' somewords.txt
someone
Someone
anyone
Anyone
$
```

A single character can be modified by a *bound*, which consists of one or two comma-separated numbers, with the first number specifying the minimum number and the second specifying the maximum. `egrep` uses curly braces (`{}`) to specify a bound, while `grep` uses back-slash curly braces (`\{\}`). These example matching strings of characters should clarify what I mean:

<code>egrep</code>	<code>grep</code>	meaning
<code>[a-z]{2,4}</code>	<code>[a-z]\{2,4\}</code>	Two through four characters
<code>[a-z]{4}</code>	<code>[a-z]\{4\}</code>	Exactly four characters
<code>[a-z]{4,}</code>	<code>[a-z]\{4,\}</code>	Four or more characters
<code>[a-z]{,4}</code>	<code>[a-z]\{,4\}</code>	Zero through four characters

Finally, the escape or backslash (`\`) removes the special meaning of a character and reverts it to a standard character. Some simple examples are illustrated below. Note that the backslash itself has a special meaning, so when you want to search for it, it must be escaped (`\\`).

character	matches
<code>.</code>	Any character
<code>\.</code>	A period
<code>\$</code>	End of line
<code>\\$</code>	A dollar sign
<code>*</code>	Zero or more occurrences of the preceding expression
<code>\*</code>	An asterisk
<code>\</code>	Nothing -- is an escape character
<code>\\</code>	A backslash
<code> </code>	Create an "or" branch between two expressions
<code> </code>	A vertical bar

The definition of the escape character dictates that, if you escape a character that doesn't need to be escaped, the escape is ignored and the character is treated as if you had entered it on its own. If you place `\a` in a search pattern, it's the same as `a`, because the letter didn't need to be escaped in the first place.

It can be hard to remember all of the `grep` and `egrep` characters that have a special meaning, and regular expressions are unfortunately far from regular. You have already seen that curly braces can be escaped in `grep` and, when escaped, acquire a special meaning. The same is true for parentheses and angle brackets. The following characters have special meanings in `grep` or `egrep`:

```
In egrep:
| ^ $ . * + ? ( ) [ { } \
In grep:
^ $ . * \( \) [ \{ \} \
```

Because regular expressions are used by `vi`, `ex`, `sed`, and `ed`, it's worth mentioning that these three editors use the following special characters:

```
^ $ . * \( \) [ \ < \>
```

As you can see, you need to be aware of the version of `grep` with which you're working before you use the backslash indiscriminately.

The last collection of `grep` or `egrep` search pattern options is in fact a simple shorthand for describing a class of characters.

```
[:alpha:] Any alphabetic character
[:lower:] Any lowercase character
[:upper:] Any uppercase character
[:digit:] Any digit
[:alnum:] Any alphanumeric character (alphabetic or digit)
[:space:] Any white space character (space, tab, vertical tab)
[:graph:] Any printable character, except space
[:print:] Any printable character, including the space
[:punct:] Any punctuation (i.e., a printable character that is not white space or alphanumeric)
[:cntrl:] Any nonprintable character
```

You may use these inside a range option. The class name includes the left and right brackets, so these must be doubled inside a range, as in the following example, which searches for any string of 10 digits. Note the apparently doubled brackets. Actually, this is an option of `[:digit:]` inside the square brackets for a range. This could also be written `[0-9]`.

```
$ egrep '[:digit:][:digit:]{10}' somenumbers.txt
1234554321
$
```

The following listing offers some example search patterns that return the line numbers containing the matches. Pattern 1 -- parentheses, followed by three digits, followed by closing parentheses, followed by three digits, a hyphen, and four digits -- searches for phone numbers.

Pattern 2 searches for zip codes -- five digits followed by zero or one hyphen, followed by zero to four digits -- either with or without the following hyphen and four digit extension.

Pattern 3 searches for lines containing P.O. Box number addresses by using a case-independent search for "p," followed by zero or one period, then zero or more spaces, zero or one period and one or more spaces, and finally "box" or "drop." This should match most of the styles of data entry for a P.O. Box, including "PO Box," "PO BOX," "P.O. Box," "P O Box," "P. O. Drop," and so on.

Pattern 4 matches the word "cat" by searching for it where it's preceded by a beginning or line, or one or more spaces and followed by one or more spaces, or an end of line. This search will not match "concatenate."

```
1. egrep -n '\([([0-9]{3})\)[0-9]{3}\-[0-9]{4}' somenumbers.txt
2. egrep -n '[0-9]{5}\-?[0-9]{0,4}' somenumbers.txt
3. egrep -in 'p\.\? *o\.\ +(box|drop)' someaddresses.txt
4. egrep -n ' (^| +)cat( +|$)' sometext.txt
```

## And, finally, fgrep

At this point you might be wondering how `fgrep` fits in with the others. `fgrep` is essentially `grep` (or `egrep`) with no special characters. If you want to search for a simple string without wild cards, use `fgrep`. The `fgrep` version of `grep` is optimized to search for strings as they appear on the command line, so it doesn't treat any characters as special. You could use `fgrep` in the above examples to more efficiently search for the plain string "hello," and also to search for strings that contain special characters used in their usual sense. For example, if you wanted to search for "hello" at the end of a sentence, you would want to search for "hello." (hello followed by a period). The dot or period is a special character in `grep` or `egrep`, but `fgrep` simply treats a period as a period and not as a special character.

```
$ fgrep 'hello.' *
```

I have two final notes about searching for multiple strings. Multiple search patterns can be placed on a single command line by using the `-e` option. The following example will search for "cat" or "dog":

```
$ fgrep -e 'cat' -e 'dog' *
```

You can also list search patterns in a file and name the file on the command line with the `-f` option. The example below is a file named `searchfor.txt` that contains a list of search patterns for the singular or plural of various animals. The question mark at the end of each animal name applies to the preceding "s" and means zero or one occurrence of that letter.

```
dogs?
cats?
ducks?
```

snakes?

To use this file to search another list of files, name it on the command line instead of a search pattern. The `egrep` utility will search for all the possible strings listed in `searchfor.txt`:

```
$ egrep -nf searchfor.txt *
```

 [Mail to a friend](#)

**[www.itworld.com](#) [open.itworld.com](#) [security.itworld.com](#) [smallbusiness.itworld.com](#)  
[storage.itworld.com](#) [utilitycomputing.itworld.com](#) [wireless.itworld.com](#)**

[Contact Us](#) [About Us](#) [Privacy Policy](#) [Terms of Service](#) [Reprints](#)

[CIO](#) [Computerworld](#) [CSO](#) [GamePro](#) [Games.net](#) [IDG Connect](#) [IDG World Expo](#) [Infoworld](#) [ITworld](#) [JavaWorld](#)  
[LinuxWorld](#) [MacUser](#) [Macworld](#) [Network World](#) [PC World](#) [Playlist](#)

Copyright © 2007 Computerworld, Inc. All rights reserved

Reproduction in whole or in part in any form or medium without express written permission of Computerworld Inc. is prohibited. Computerworld and Computerworld.com and the respective logos are trademarks of International Data Group Inc.