Searching for Files

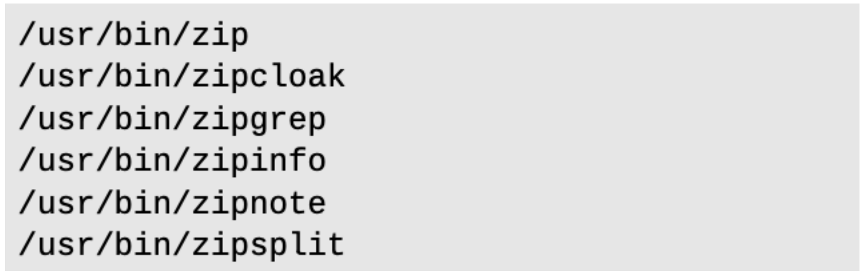
- `locate` – Find files by name
- `find` – Search for files in a directory hierarchy
- `xargs` – Build and execute command lines from standard input
- `touch` – Change file times
- `stat` – Display file or file system status

`locate` – Find Files the Easy Way

- The locate program performs a rapid database search of pathnames, and then outputs every name that matches a given substring.
- Say, for example, we want to find all the programs with names that begin with `zip`. Since we are looking for programs, we can assume that the name of the directory containing the programs would end with `bin/`.
- Therefore, we could try to use locate this way to find our files:

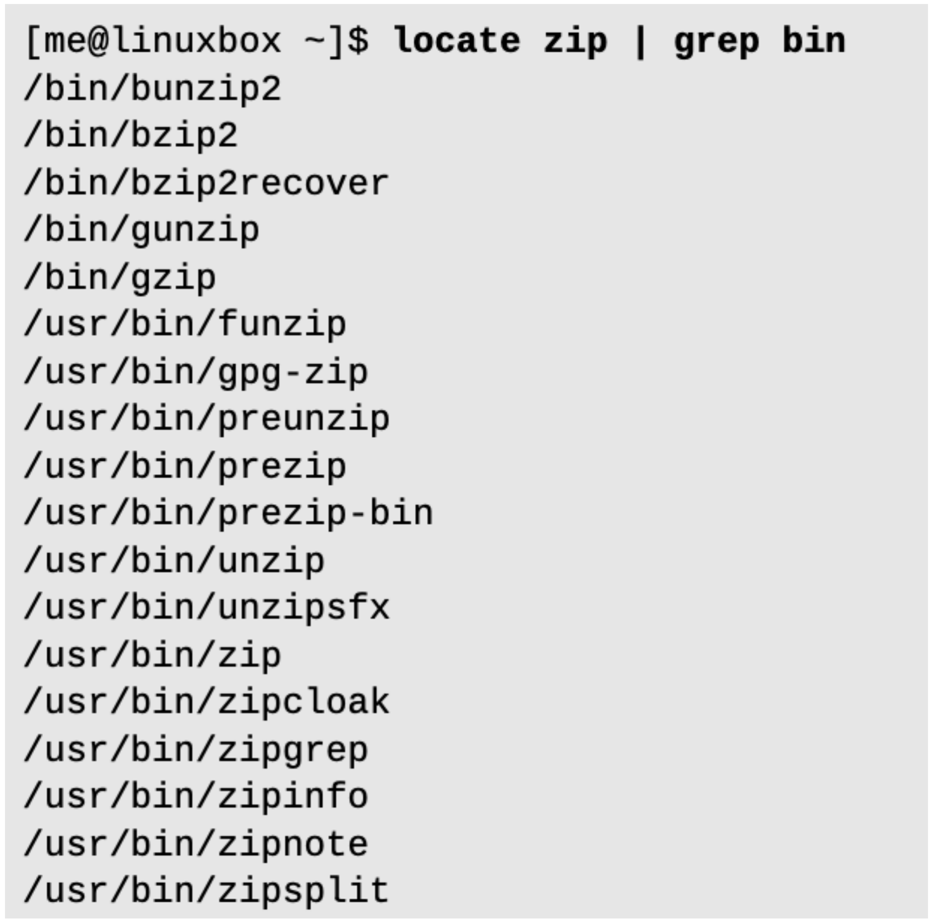
```
locate bin/zip
```

- locate will search its database of pathnames and output any that contain the string `bin/zip`.



```
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

- If the search requirement is not so simple, we can combine locate with other tools such as `grep` to design more interesting searches.



```
[me@linuxbox ~]$ locate zip | grep bin
/bin/bunzip2
/bin/bzip2
/bin/bzip2recover
/bin/gunzip
/bin/gzip
/usr/bin/funzip
/usr/bin/gpg-zip
/usr/bin/preunzip
/usr/bin/prezip
/usr/bin/prezip-bin
/usr/bin/unzip
/usr/bin/unzipsfx
/usr/bin/zip
/usr/bin/zipcloak
/usr/bin/zipgrep
/usr/bin/zipinfo
/usr/bin/zipnote
/usr/bin/zipsplit
```

find – Find Files the Hard Way

- While the locate program can find a file based solely on its name, the find program searches a given directory (and its subdirectories) for files based on a variety of attributes.
- We’re going to spend a lot of time with find because it has a lot of interesting features that we will see again and again when we start to cover programming concepts in later chapters.
- In its simplest use, find is given one or more names of directories to search. For example, to produce a listing of our home directory we can use this:

```
find ~
```

- On most active user accounts, this will produce a large list. Since the list is sent to standard output, we can pipe the list into other programs. Let’s use wc to count the number of files.

```
find ~ | wc -l
47068
```

Tests

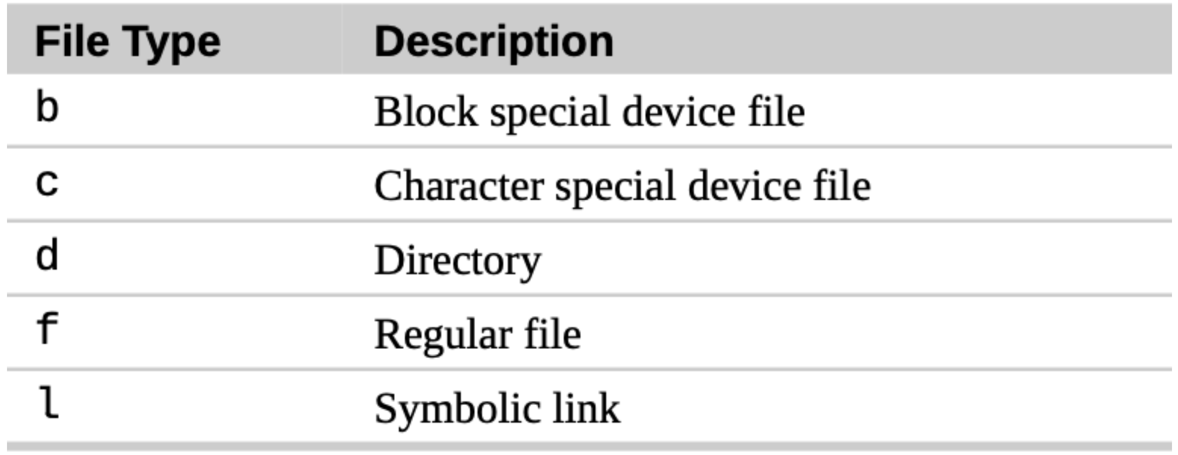
- Let’s say we want a list of directories from our search. To do this, we could add the following test:

```
find ~ -type d | wc -l
1695
```

- Adding the test -type d limited the search to directories. Conversely, we could have limited the search to regular files with this test:

```
find ~ -type f | wc -l
38737
```

- The table lists the common file type tests supported by find.

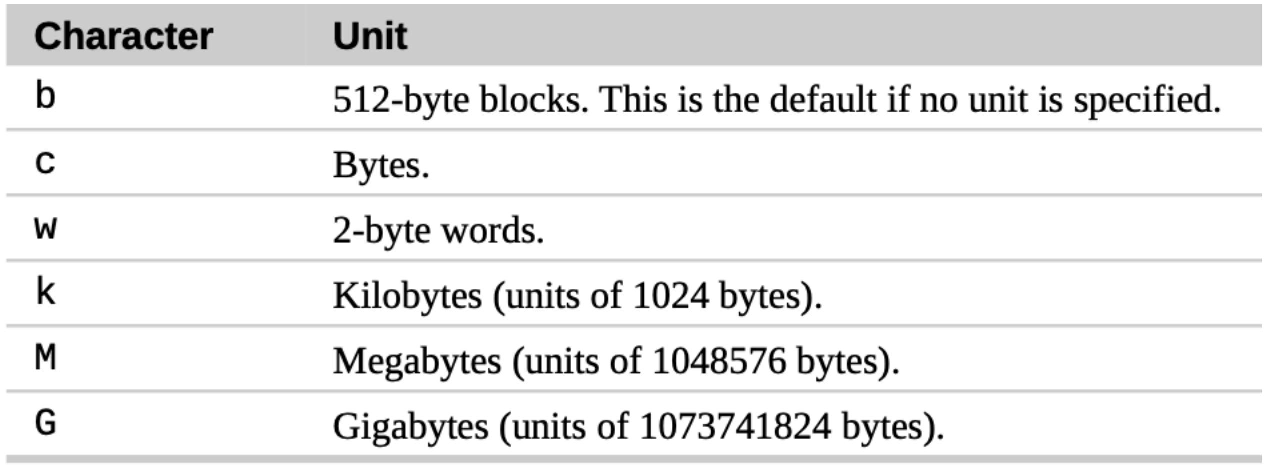


File Type	Description
b	Block special device file
c	Character special device file
d	Directory
f	Regular file
l	Symbolic link

- We can also search by file size and filename by adding some additional tests. Let’s look for all the regular files that match the wildcard pattern *.JPG and are larger than one megabyte.

```
find ~ -type f -name "*.JPG" -size +1M | wc -l
840
```

- In this example, we add the -name test followed by the wildcard pattern. Notice how we enclose it in quotes to prevent pathname expansion by the shell. Next, we add the –size test followed by the string “+1M”.
- The leading plus sign indicates that we are looking for files larger than the specified number. A leading minus sign would change the meaning of the string to be smaller than the specified number. Using no sign means, “match the value exactly.”
- The trailing letter M indicates that the unit of measurement is megabytes. The table lists the characters that can be used to specify units.



Character	Unit
b	512-byte blocks. This is the default if no unit is specified.
c	Bytes.
w	2-byte words.
k	Kilobytes (units of 1024 bytes).
M	Megabytes (units of 1048576 bytes).
G	Gigabytes (units of 1073741824 bytes).

- `find` supports a large number of tests. The table provides a rundown of the common ones.
- Note that in cases where a numeric argument is required, the same + and – notation discussed above can be applied.

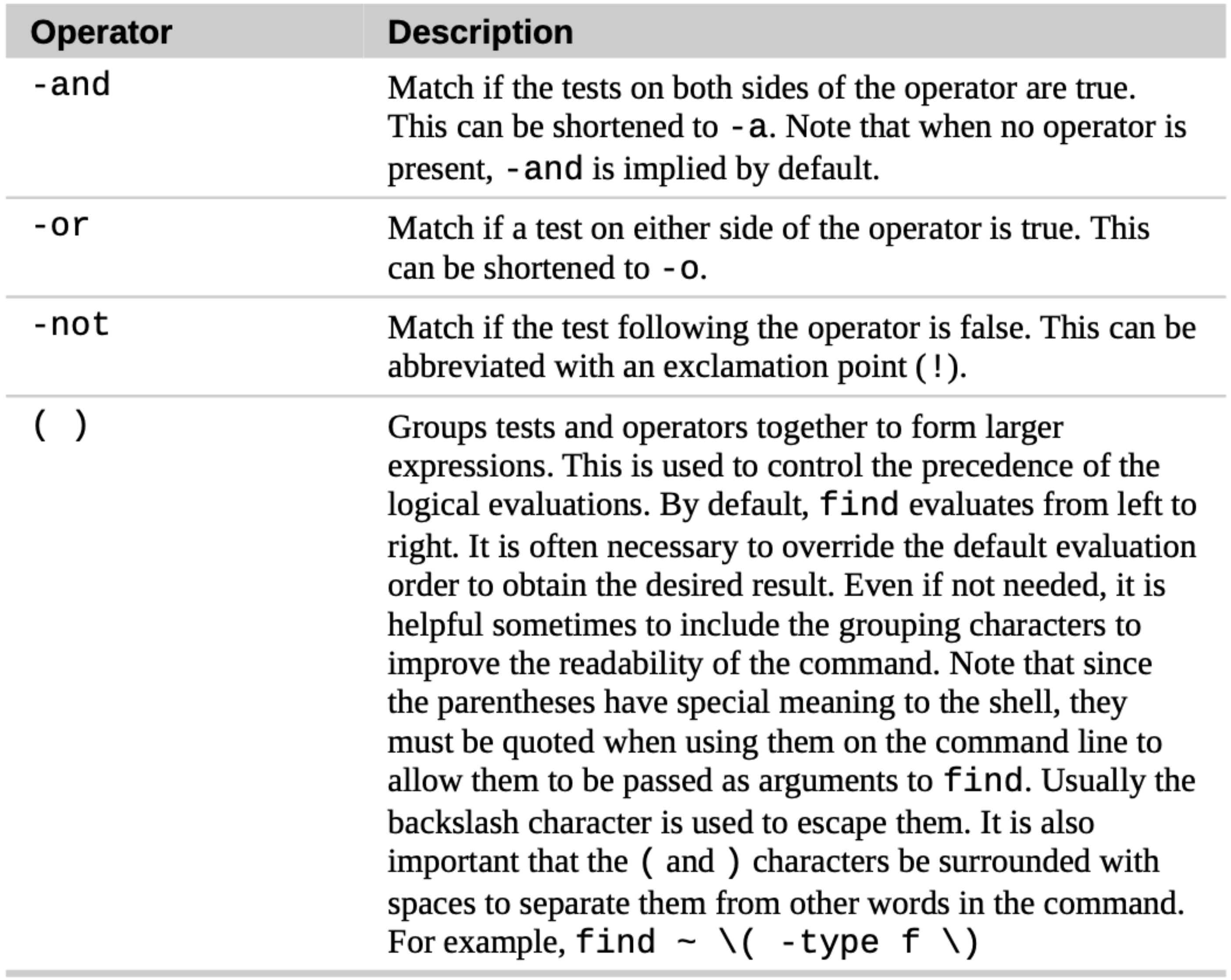
Test	Description
`-cmin` *`n`*	Match files or directories whose content or attribu last modified exactly *n* minutes ago. To specify le minutes ago, use `-`*`n`*, and to specify more than *n* n ago, use `+`*`n`*.
`-cnewer` *`file`*	Match files or directories whose contents or attrib last modified more recently than those of *`file`*.
`-ctime` *`n`*	Match files or directories whose contents or attribu as ownership or permissions) were last modified *n* hours ago.
`-empty`	Match empty files and directories.
`-group` *`name`*	Match file or directories belonging to *`group`*. *`gr`* be expressed either as a group name or as a numer ID.
`-iname` *`pattern`*	Like the `-name` test but case-insensitive.
`-inum` *`n`*	Match files with inode number *n*. This is helpful for finding all the hard links to a particular inode.
`-mmin` *`n`*	Match files or directories whose contents were last modified *n* minutes ago.
`-mtime` *`n`*	Match files or directories whose contents were last modified *n**24 hours ago.
`-name` *`pattern`*	Match files and directories with the specified wildcard *`pattern`*.
`-newer` *`file`*	Match files and directories whose contents were modified more recently than the specified *`file`*. This is useful when writing shell scripts that perform file backups. Each time you make a backup, update a file (such as a log) and then use `find` to determine which files have changed since the last update.
`-nouser`	Match file and directories that do not belong to a valid user. This can be used to find files belonging to deleted accounts or to detect activity by attackers.
`-nogroup`	Match files and directories that do not belong to a valid group.
`-perm` *`mode`*	Match files or directories that have permissions set to the specified *`mode`*. *`mode`* can be expressed by either octal or symbolic notation.
`-samefile` *`name`*	Similar to the `-inum` test. Match files that share the same inode number as file *`name`*.
`-size` *`n`*	Match files of size *n*.
`-type` *`c`*	Match files of type *`c`*.
`-user` *`name`*	Match files or directories belonging to user *`name`*. The user may be expressed by a username or by a numeric user ID.

Operators

- Even with all the tests that find provides, we may still need a better way to describe the logical relationships between the tests.
- For example, what if we needed to determine whether all the files and subdirectories in a directory had secure permissions?
- We would look for all the files with permissions that are not 0600 and the directories with permissions that are not 0700.
- Fortunately, find provides a way to combine tests using logical operators to create more complex logical relationships. To express the aforementioned test, we could do this:

```
find ~ \( -type f -not -perm 0600 \) -or \( -type d -not
-perm 0700 \)
```

- The table describes the logical operators used with find.



Operator	Description
`-and`	Match if the tests on both sides of the operator are true. This can be shortened to `-a`. Note that when no operator is present, `-and` is implied by default.
`-or`	Match if a test on either side of the operator is true. This can be shortened to `-o`.
`-not`	Match if the test following the operator is false. This can be abbreviated with an exclamation point (`!`).
`( )`	Groups tests and operators together to form larger expressions. This is used to control the precedence of the logical evaluations. By default, `find` evaluates from left to right. It is often necessary to override the default evaluation order to obtain the desired result. Even if not needed, it is helpful sometimes to include the grouping characters to improve the readability of the command. Note that since the parentheses have special meaning to the shell, they must be quoted when using them on the command line to allow them to be passed as arguments to `find`. Usually the backslash character is used to escape them. It is also important that the `(` and `)` characters be surrounded with spaces to separate them from other words in the command. For example, `find ~ \( -type f \)`

- With this list of operators in hand, let’s deconstruct our find command. When viewed from the uppermost level, we see that our tests are arranged as two groupings separated by an -or operator.

```
( expression 1 ) -or ( expression 2 )
```

- This makes sense, since we are searching for files with a certain set of permissions and for directories with a different set. If we are looking for both files and directories, why do we use -or instead of -and? As find scans through the files and directories, each one is evaluated to see whether it matches the specified tests. We want to know whether it is either a file with bad permissions or a directory with bad permissions. It can’t be both at the same time. So if we expand the grouped expressions, we can see it this way:

```
( file with bad perms ) -or ( directory with bad perms )
```

- Our next challenge is how to test for “bad permissions.” How do we do that? Actually, we don’t. What we will test for is “not good permissions” since we know what “good per- missions” are. In the case of files, we define good as 0600 and for directories, we define it as 0700. The expression that will test files for “not good” permissions is as follows:

```
-type f -and -not -perm 0600
```

- For directories it is as follows:

```
-type d -and -not -perm 0700
```

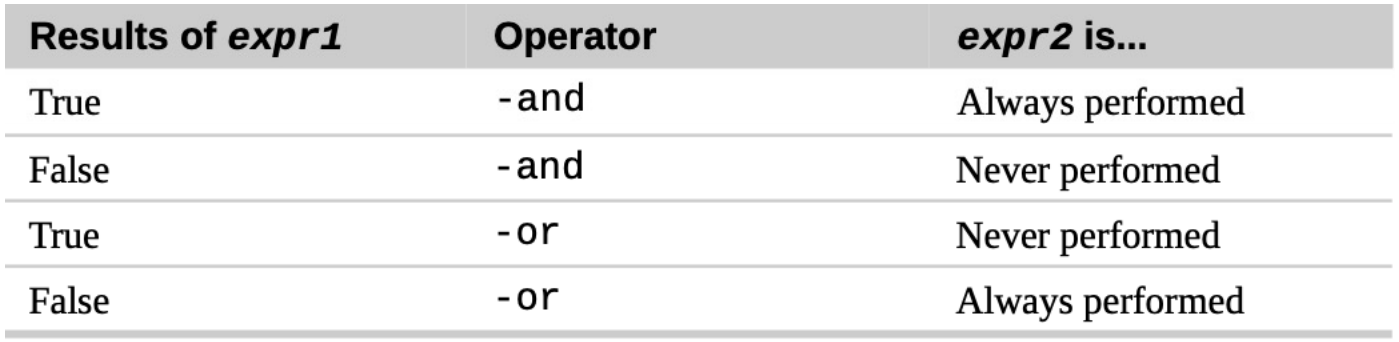
- As noted in the Table above, the -and operator can be safely removed since it is implied by default. So if we put this all back together, we get our final command.

```
find ~ ( -type f -not -perm 0600 ) -or ( -type d -not –perm 0700 )
```

- However, since the parentheses have special meaning to the shell, we must escape them to prevent the shell from trying to interpret them. Preceding each one with a backslash character does the trick.
- There is another feature of logical operators that is important to understand. Let’s say that we have two expressions separated by a logical operator.

```
expr1 -operator expr2
```

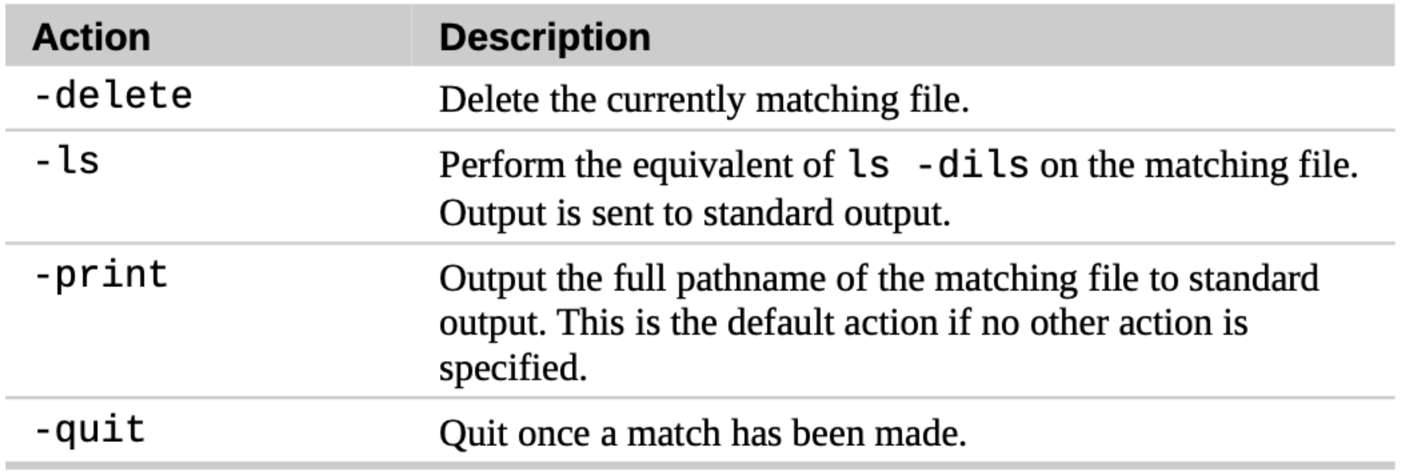
- In all cases, expr1 will always be performed; however, the operator will determine whether expr2 is performed.
- The table below outlines how it works.



Results of ***expr1***	**Operator**	***expr2* is...**
True	`-and`	Always performed
False	`-and`	Never performed
True	`-or`	Never performed
False	`-or`	Always performed

Predefined Actions

- Having a list of results from our find command is useful, but what we really want to do is act on the items on the list.
- Fortunately, find allows actions to be performed based on the search results.
- There are a set of predefined actions and several ways to apply user-defined actions. First, let’s look at a few of the predefined actions listed in the table below.



Action	Description
`-delete`	Delete the currently matching file.
`-ls`	Perform the equivalent of `ls -dils` on the matching file. Output is sent to standard output.
`-print`	Output the full pathname of the matching file to standard output. This is the default action if no other action is specified.
`-quit`	Quit once a match has been made.

- As with the tests, there are many more actions. See the find man page for full details.
- In the first example, we did this:

```
find ~
```

- This produced a list of every file and subdirectory contained within our home directory. It produced a list because the -print action is implied if no other action is specified.
- Thus, our command could also be expressed as follows:

```
find ~ -print
```

- We can use find to delete files that meet certain criteria. For example, to delete files that have the file extension .bak (which is often used to designate backup files), we could use this command:

```
find ~ -type f -name '*.bak' -delete
```

- In this example, every file in the user’s home directory (and its subdirectories) is searched for filenames ending in .bak. When they are found, they are deleted.

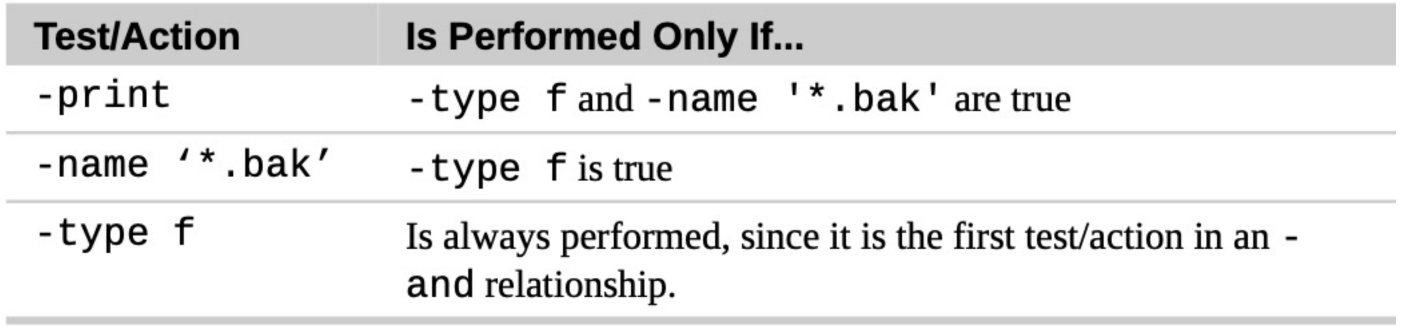
- let’s take another look at how the logical operators affect actions. Consider the following command:

```
find ~ -type f -name '*.bak' -print
```

- As we have seen, this command will look for every regular file (-type f) whose name ends with .bak (-name '*.bak') and will output the relative pathname of each matching file to standard output (-print).
- However, the reason the command performs the way it does is determined by the logical relationships between each of the tests and actions. Remember, there is, by default, an implied -and relationship between each test and action. We could also express the command this way to make the logical relationships easier to see:

```
find ~ -type f -and -name '*.bak' -and -print
```

- With our command fully expressed, let’s look at how the logical operators affect its execution:



Test/Action	Is Performed Only If...
`-print`	`-type f` and `-name '*.bak'` are true
`-name ‘*.bak’`	`-type f` is true
`-type f`	Is always performed, since it is the first test/action in an `-and` relationship.

- Since the logical relationship between the tests and actions determines which of them are performed, we can see that the order of the tests and actions is important. For instance, if we were to reorder the tests and actions so that the -print action was the first one, the command would behave much differently.

```
find ~ -print -and -type f -and -name '*.bak'
```

- This version of the command will print each file (the -print action always evaluates to true) and then test for file type and the specified file extension.

User-Defined Actions

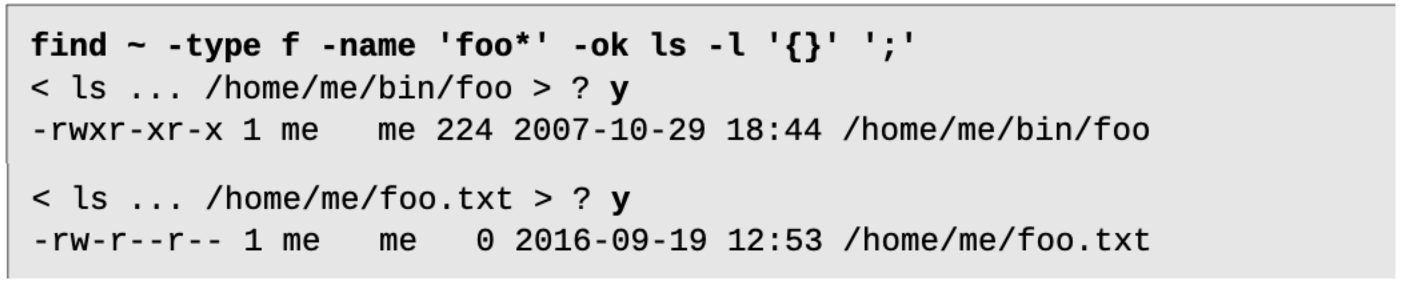
- In addition to the predefined actions, we can also invoke arbitrary commands. The traditional way of doing this is with the -exec action. This action works like this:

  ```
  -exec command {} ;
  ```

- Here command is the name of a command, {} is a symbolic representation of the current pathname, and the semicolon is a required delimiter indicating the end of the command.
- Here's an example of using -exec to act like the -delete action discussed earlier:

  ```
  -exec rm '{}' ';'
  ```

- Again, since the brace and semicolon characters have special meaning to the shell, they must be quoted or escaped.
- It's also possible to execute a user-defined action interactively. By using the -ok action in place of -exec, the user is prompted before execution of each specified command.



```
find ~ -type f -name 'foo*' -ok ls -l '{}' ';'
< ls ... /home/me/bin/foo > ? y
-rwxr-xr-x 1 me   me 224 2007-10-29 18:44 /home/me/bin/foo

< ls ... /home/me/foo.txt > ? y
-rw-r--r-- 1 me   me   0 2016-09-19 12:53 /home/me/foo.txt
```

- In this example, we search for files with names starting with the string foo and execute the command ls -l each time one is found. Using the -ok action prompts the user be- fore the ls command is executed.

Improving Efficiency

- When the -exec action is used, it launches a new instance of the specified command each time a matching file is found. There are times when we might prefer to combine all of the search results and launch a single instance of the command. For example, rather than executing the commands like this:

```
ls -l file1
ls -l file2
```

- we may prefer to execute them this way:

```
ls -l file1 file2
```

- This causes the command to be executed only one time rather than multiple times. There are two ways we can do this: the traditional way, using the external command `xargs` and the alternate way, using a new feature in find itself.

- By changing the trailing semicolon character to a plus sign, we activate the ability of find to combine the results of the search into an argument list for a single execution of the desired command.
- Going back to our example, this will execute ls each time a matching file is found:

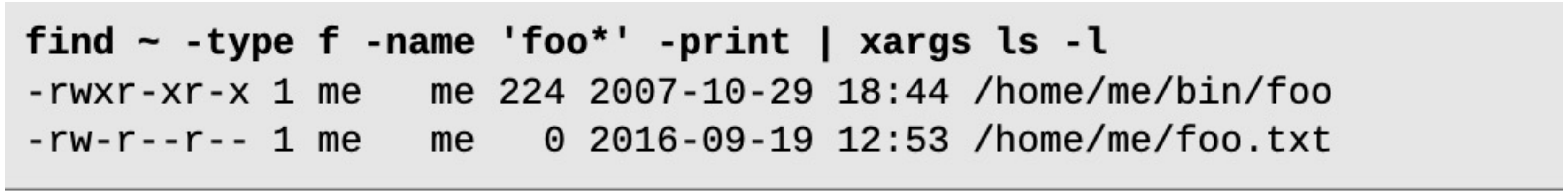
```
find ~ -type f -name 'foo*' -exec ls -l '{}' ';'
-rwxr-xr-x 1 me   me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me   me   0 2016-09-19 12:53 /home/me/foo.txt
```

- By changing the command to the following, we get the same results, but the system has to execute the ls command only once.

```
find ~ -type f -name 'foo*' -exec ls -l '{}' +
-rwxr-xr-x 1 me   me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me   me   0 2016-09-19 12:53 /home/me/foo.txt
```

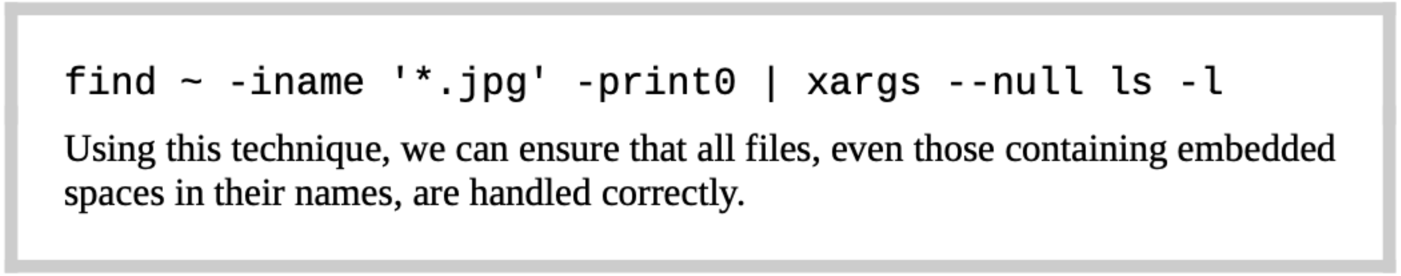
xargs

- The xargs command performs an interesting function. It accepts input from standard input and converts it into an argument list for a specified command. With our example, we would use it like this:



```
find ~ -type f -name 'foo*' -print | xargs ls -l
-rwxr-xr-x 1 me   me 224 2007-10-29 18:44 /home/me/bin/foo
-rw-r--r-- 1 me   me   0 2016-09-19 12:53 /home/me/foo.txt
```

- Here we see the output of the find command piped into xargs, which, in turn, constructs an argument list for the ls command and then executes it.



```
find ~ -iname '*.jpg' -print0 | xargs --null ls -l
```

Using this technique, we can ensure that all files, even those containing embedded spaces in their names, are handled correctly.

A Return to the Playground

- It’s time to put find to some (almost) practical use. We’ll create a playground and try some of what we have learned.
- First, let’s create a playground with lots of subdirectories and files.

```
mkdir -p playground/dir-{001..100}
touch playground/dir-{001..100}/file-{A..Z}
```

- With these two lines, we created a playground directory containing 100 subdirectories each containing 26 empty files.

- By combining mkdir with the -p option (which causes mkdir to create the parent directories of the specified paths) with brace expansion, we were able to create 100 subdirectories.
- The touch command is usually used to set or update the access, change, and modify times of files. However, if a filename argument is that of a non-existent file, an empty file is created.
- In our playground, we created 100 instances of a file named file-A. Let’s find them.

```
find playground -type f -name 'file-A'
```

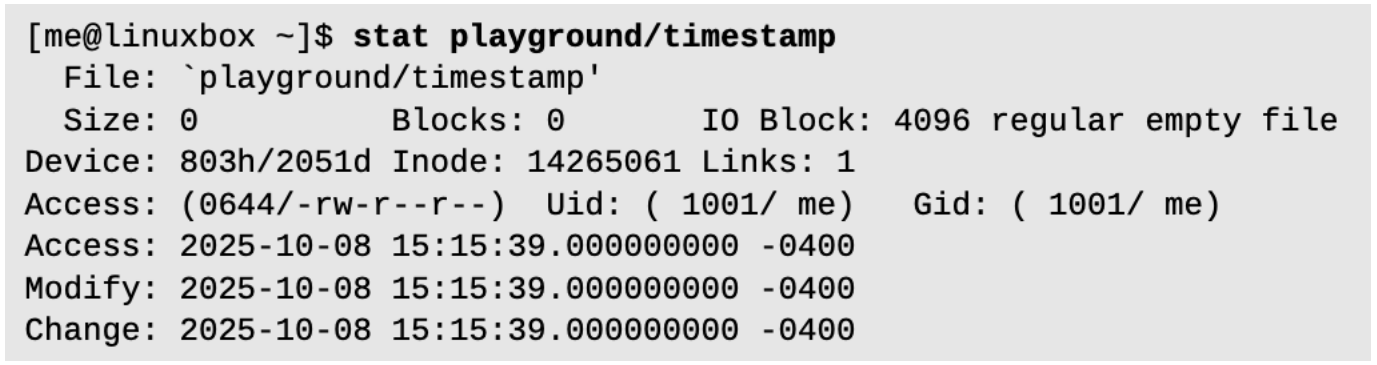
- Note that unlike ls, find does not produce results in sorted order. Its order is determined by the layout of the storage device. We can confirm that we actually have 100 instances of the file this way.

```
[me@linuxbox ~]$ find playground -type f -name 'file-A' | wc -l
100
```

- let’s look at finding files based on their modification times. This will be helpful when creating backups or organizing files in chronological order. To do this, we will first create a reference file against which we will compare modification time.

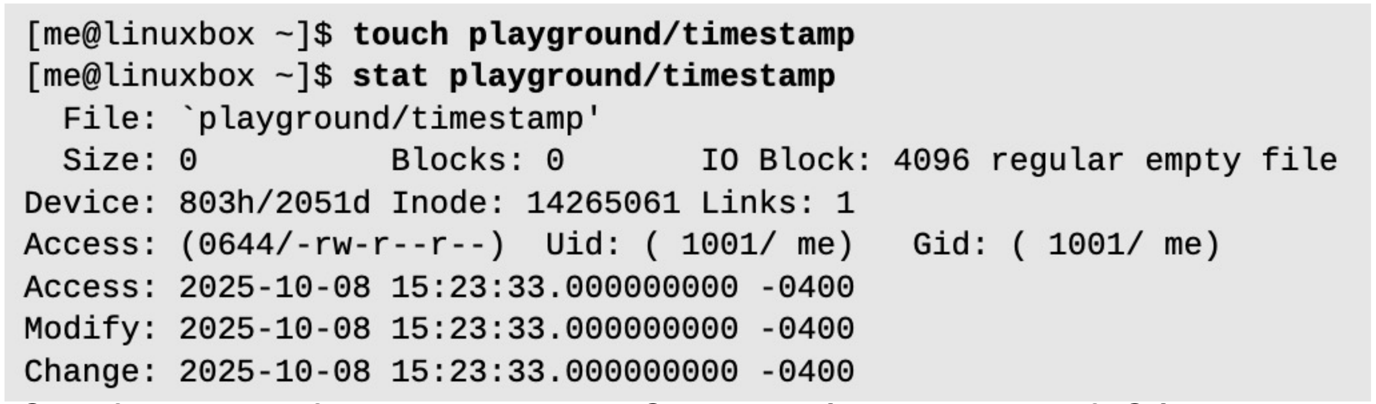
```
touch playground/timestamp
```

- This creates an empty file named timestamp and sets its modification time to the current time.
- We can verify this by using another handy command, stat, which is a kind of souped-up version of `ls`.
- The `stat` command reveals all that the system understands about a file and its attributes.



```
[me@linuxbox ~]$ stat playground/timestamp
  File: `playground/timestamp'
  Size: 0          Blocks: 0       IO Block: 4096 regular empty file
Device: 803h/2051d Inode: 14265061 Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1001/ me)   Gid: ( 1001/ me)
Access: 2025-10-08 15:15:39.000000000 -0400
Modify: 2025-10-08 15:15:39.000000000 -0400
Change: 2025-10-08 15:15:39.000000000 -0400
```

- If we use touch again and then examine the file with stat, we will see that the file’s times have been updated.



```
[me@linuxbox ~]$ touch playground/timestamp
[me@linuxbox ~]$ stat playground/timestamp
  File: `playground/timestamp'
  Size: 0          Blocks: 0       IO Block: 4096 regular empty file
Device: 803h/2051d Inode: 14265061 Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1001/ me)   Gid: ( 1001/ me)
Access: 2025-10-08 15:23:33.000000000 -0400
Modify: 2025-10-08 15:23:33.000000000 -0400
Change: 2025-10-08 15:23:33.000000000 -0400
```

- Let’s use find to update some of our playground files.

```
find playground -type f -name 'file-B' -exec touch '{}' ';'
```

- This updates all files in the playground named file-B. Next we’ll use find to identify the updated files by comparing all the files to the reference file timestamp.

```
find playground -type f -newer playground/timestamp
```

- The results contain all 100 instances of file-B. Since we performed a touch on all the files in the playground named file-B after we updated timestamp, they are now “newer” than timestamp and thus can be identified with the `-newer` test.

- let’s go back to the bad permissions test we performed earlier and apply it to playground.

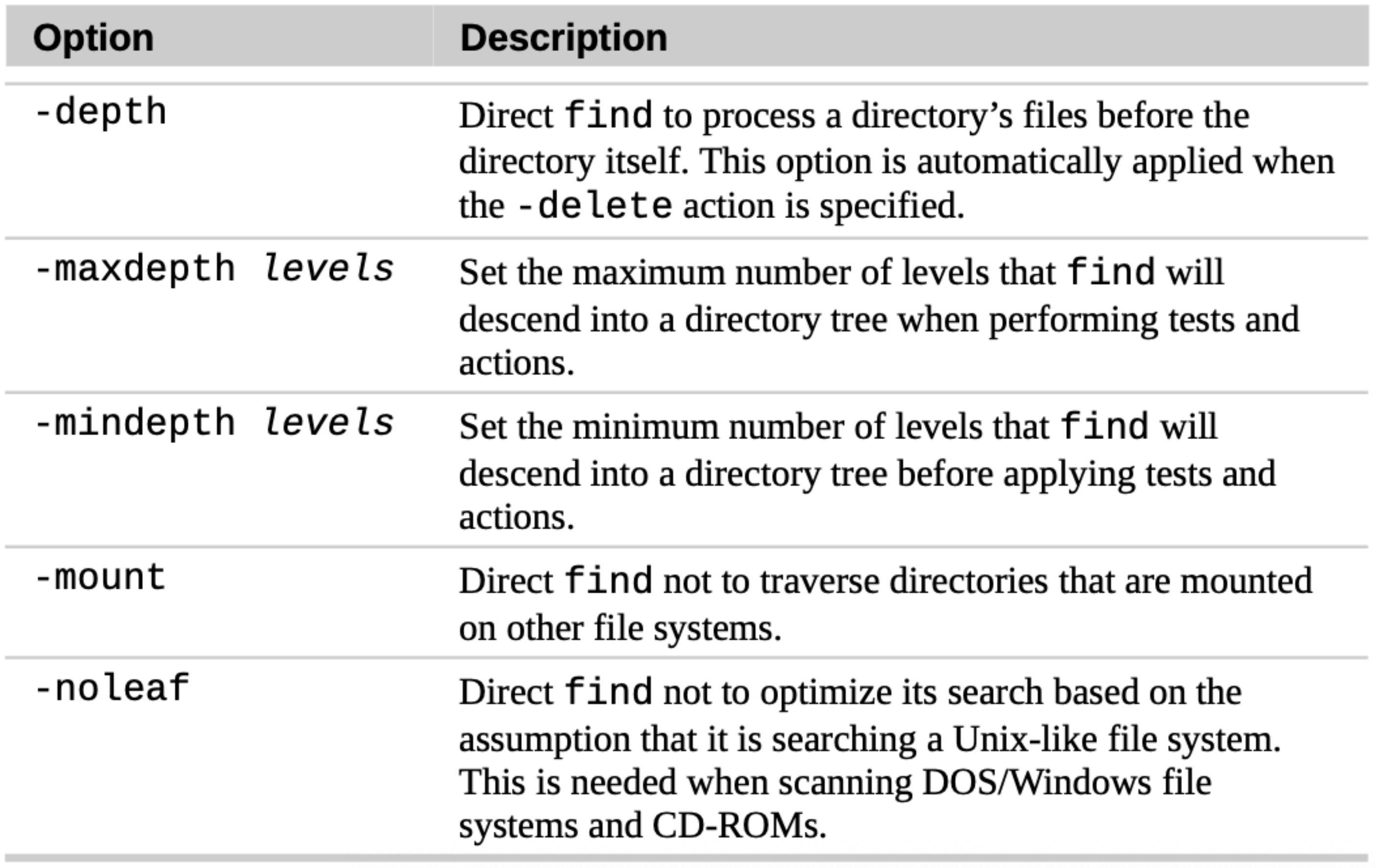
```
find playground \( -type f -not -perm 0600 \) -or \( -type
d -not -perm 0700 \)
```

- This command lists all 100 directories and 2,600 files in playground (as well as timestamp and playground itself, for a total of 2,702) because none of them meets our definition of “good permissions.” With our knowledge of operators and actions, we can add actions to this command to apply new permissions to the files and directories in our playground.

```
find playground \( -type f -not -perm 0600 –exec chmod 0600 '{}' ';'
\) -or \( -type d -not -perm 0700 -exec chmod 0700 '{}' ';' \)
```

Options

- The options are used to control the scope of a find search. They may be included with other tests and actions when constructing find expressions. The table lists the most commonly used find options.



Option	Description
`-depth`	Direct `find` to process a directory's files before the directory itself. This option is automatically applied when the `-delete` action is specified.
`-maxdepth` *`levels`*	Set the maximum number of levels that `find` will descend into a directory tree when performing tests and actions.
`-mindepth` *`levels`*	Set the minimum number of levels that `find` will descend into a directory tree before applying tests and actions.
`-mount`	Direct `find` not to traverse directories that are mounted on other file systems.
`-noleaf`	Direct `find` not to optimize its search based on the assumption that it is searching a Unix-like file system. This is needed when scanning DOS/Windows file systems and CD-ROMs.