Finding Files

Edition 4.2.20, for GNU `find` version 4.2.20
2 March 2005

by David MacKenzie

This file documents the GNU utilities for finding files that match certain criteria and performing various operations on them.

Short Contents

Table of Contents

1 Introduction

This manual shows how to find files that meet criteria you specify, and how to perform various actions on the files that you find. The principal programs that you use to perform these tasks are `find`, `locate`, and `xargs`. Some of the examples in this manual use capabilities specific to the GNU versions of those programs.

GNU `find` was originally written by Eric Decker, with enhancements by David MacKenzie, Jay Plett, and Tim Wood. GNU `xargs` was originally written by Mike Rendell, with enhancements by David MacKenzie. GNU `locate` and its associated utilities were originally written by James Woods, with enhancements by David MacKenzie. The idea for '`find -print0`' and '`xargs -0`' came from Dan Bernstein. The current maintainer of GNU findutils (and this manual) is James Youngman. Many other people have contributed bug fixes, small improvements, and helpful suggestions. Thanks!

Mail suggestions and bug reports for these programs to `bug-findutils@gnu.org`. Please include the version number, which you can get by running '`find --version`'.

1.1 Scope

For brevity, the word *file* in this manual means a regular file, a directory, a symbolic link, or any other kind of node that has a directory entry. A directory entry is also called a *file name*. A file name may contain some, all, or none of the directories in a path that leads to the file. These are all examples of what this manual calls "file names":

```
parser.c
README
./budget/may-94.sc
fred/.cshrc
/usr/local/include/termcap.h
```

A *directory tree* is a directory and the files it contains, all of its subdirectories and the files they contain, etc. It can also be a single non-directory file.

These programs enable you to find the files in one or more directory trees that:

- have names that contain certain text or match a certain pattern;
- are links to certain files;
- were last used during a certain period of time;
- are within a certain size range;
- are of a certain type (regular file, directory, symbolic link, etc.);
- are owned by a certain user or group;
- have certain access permissions;
- contain text that matches a certain pattern;
- are within a certain depth in the directory tree;
- or some combination of the above.

Once you have found the files you're looking for (or files that are potentially the ones you're looking for), you can do more to them than simply list their names. You can get any combination of the files' attributes, or process the files in many ways, either individually or in groups of various sizes. Actions that you might want to perform on the files you have found include, but are not limited to:

- view or edit
- store in an archive
- remove or rename

- change access permissions
- classify into groups

This manual describes how to perform each of those tasks, and more.

1.2 Overview

The principal programs used for making lists of files that match given criteria and running commands on them are `find`, `locate`, and `xargs`. An additional command, `updatedb`, is used by system administrators to create databases for `locate` to use.

`find` searches for files in a directory hierarchy and prints information about the files it found. It is run like this:

```
find [file...] [expression]
```

Here is a typical use of `find`. This example prints the names of all files in the directory tree rooted in '`/usr/src`' whose name ends with '`.c`' and that are larger than 100 Kilobytes.

```
find /usr/src -name '*.c' -size +100k -print
```

`locate` searches special file name databases for file names that match patterns. The system administrator runs the `updatedb` program to create the databases. `locate` is run like this:

```
locate [option...] pattern...
```

This example prints the names of all files in the default file name database whose name ends with '`Makefile`' or '`makefile`'. Which file names are stored in the database depends on how the system administrator ran `updatedb`.

```
locate '*[Mm]akefile'
```

The name `xargs`, pronounced EX-args, means "combine arguments." `xargs` builds and executes command lines by gathering together arguments it reads on the standard input. Most often, these arguments are lists of file names generated by `find`. `xargs` is run like this:

```
xargs [option...] [command [initial-arguments]]
```

The following command searches the files listed in the file '`file-list`' and prints all of the lines in them that contain the word '`typedef`'.

```
xargs grep typedef < file-list
```

1.3 `find` Expressions

The expression that `find` uses to select files consists of one or more *primaries*, each of which is a separate command line argument to `find`. `find` evaluates the expression each time it processes a file. An expression can contain any of the following types of primaries:

options	affect overall operation rather than the processing of a specific file;
tests	return a true or false value, depending on the file's attributes;
actions	have side effects and return a true or false value; and
operators	connect the other arguments and affect when and whether they are evaluated.

You can omit the operator between two primaries; it defaults to '`-and`'. See Section 2.11 [Combining Primaries With Operators], page 13, for ways to connect primaries into more complex expressions. If the expression contains no actions other than '`-prune`', '`-print`' is performed on all files for which the entire expression is true (see Section 3.1 [Print File Name], page 15).

Options take effect immediately, rather than being evaluated for each file when their place in the expression is reached. Therefore, for clarity, it is best to place them at the beginning of the expression.

Many of the primaries take arguments, which immediately follow them in the next command line argument to `find`. Some arguments are file names, patterns, or other strings; others are numbers. Numeric arguments can be specified as

`+n`	for greater than *n*,
`-n`	for less than *n*,
`n`	for exactly *n*.

2 Finding Files

By default, `find` prints to the standard output the names of the files that match the given criteria. See Chapter 3 [Actions], page 15, for how to get more information about the matching files.

2.1 Name

Here are ways to search for files whose name matches a certain pattern. See Section 2.1.4 [Shell Pattern Matching], page 5, for a description of the *pattern* arguments to these tests.

Each of these tests has a case-sensitive version and a case-insensitive version, whose name begins with '`i`'. In a case-insensitive comparison, the patterns '`fo*`' and '`F??`' match the file names '`Foo`', '`FOO`', '`foo`', '`fOo`', etc.

2.1.1 Base Name Patterns

`-name` *pattern* [Test]
`-iname` *pattern* [Test]

True if the base of the file name (the path with the leading directories removed) matches shell pattern *pattern*. For '`-iname`', the match is case-insensitive. To ignore a whole directory tree, use '`-prune`' (see Section 2.9 [Directories], page 12). As an example, to find Texinfo source files in '`/usr/local/doc`':

```
find /usr/local/doc -name '*.texi'
```

Patterns for '`-name`' and '`-iname`' will match a filename with a leading '`.`'. For example the command '`find /tmp -name \*bar`' will match the file '`/tmp/.foobar`'.

2.1.2 Full Name Patterns

`-wholename` *pattern* [Test]
`-iwholename` *pattern* [Test]

True if the entire file name, starting with the command line argument under which the file was found, matches shell pattern *pattern*. For '`-iwholename`', the match is case-insensitive. To ignore a whole directory tree, use '`-prune`' rather than checking every file in the tree (see Section 2.9 [Directories], page 12). The "entire file name" as used by find starts with the starting-point specified on the command line, and is not converted to an absolute pathname, so for example `cd /; find tmp -wholename /tmp` will never match anything.

`-path` *pattern* [Test]
`-ipath` *pattern* [Test]

These tests are deprecated, but work as for '`-wholename`' and '`-iwholename`', respectively. The '`-ipath`' test is a GNU extension, but '`-path`' is also provided by HP-UX `find`.

`-regex` *expr* [Test]
`-iregex` *expr* [Test]

True if the entire file name matches regular expression *expr*. This is a match on the whole path, not a search. For example, to match a file named '`./fubar3`', you can use the regular expression '`.*bar.`' or '`.*b.*3`', but not '`f.*r3`'. See section "Syntax of Regular Expressions" in *The GNU Emacs Manual*, for a description of the syntax of regular expressions. For '`-iregex`', the match is case-insensitive.

2.1.3 Fast Full Name Search

To search for files by name without having to actually scan the directories on the disk (which can be slow), you can use the `locate` program. For each shell pattern you give it, `locate` searches one or more databases of file names and displays the file names that contain the pattern. See Section 2.1.4 [Shell Pattern Matching], page 5, for details about shell patterns.

If a pattern is a plain string—it contains no metacharacters—`locate` displays all file names in the database that contain that string. If a pattern contains metacharacters, `locate` only displays file names that match the pattern exactly. As a result, patterns that contain metacharacters should usually begin with a '`*`', and will most often end with one as well. The exceptions are patterns that are intended to explicitly match the beginning or end of a file name.

If you only want `locate` to match against the last component of the filenames (the "base name" of the files) you can use the '`--basename`' option. The opposite behaviour is the default, but can be selected explicitly by using the option '`--wholename`'.

The command

```
locate pattern
```

is almost equivalent to

```
find directories -name pattern
```

where *directories* are the directories for which the file name databases contain information. The differences are that the `locate` information might be out of date, and that `locate` handles wildcards in the pattern slightly differently than `find` (see Section 2.1.4 [Shell Pattern Matching], page 5).

The file name databases contain lists of files that were on the system when the databases were last updated. The system administrator can choose the file name of the default database, the frequency with which the databases are updated, and the directories for which they contain entries.

Here is how to select which file name databases `locate` searches. The default is system-dependent.

`--database=`*path*
`-d` *path* Instead of searching the default file name database, search the file name databases in *path*, which is a colon-separated list of database file names. You can also use the environment variable `LOCATE_PATH` to set the list of database files to search. The option overrides the environment variable if both are used.

2.1.4 Shell Pattern Matching

`find` and `locate` can compare file names, or parts of file names, to shell patterns. A *shell pattern* is a string that may contain the following special characters, which are known as *wildcards* or *metacharacters*.

You must quote patterns that contain metacharacters to prevent the shell from expanding them itself. Double and single quotes both work; so does escaping with a backslash.

`*` Matches any zero or more characters.

`?` Matches any one character.

`[`*string*`]` Matches exactly one character that is a member of the string *string*. This is called a *character class*. As a shorthand, *string* may contain ranges, which consist of two characters with a dash between them. For example, the class '`[a-z0-9_]`' matches a lowercase letter, a number, or an underscore. You can negate a class by placing a '`!`' or '`^`' immediately after the opening bracket. Thus, '`[^A-Z@]`' matches any character except an uppercase letter or an at sign.

`\` Removes the special meaning of the character that follows it. This works even in character classes.

In the `find` tests that do shell pattern matching ('`-name`', '`-wholename`', etc.), wildcards in the pattern will match a '`.`' at the beginning of a file name. This is also the case for `locate`. Thus, '`find -name '*macs'`' will match a file named '`.emacs`', as will '`locate '*macs'`'.

Slash characters have no special significance in the shell pattern matching that `find` and `locate` do, unlike in the shell, in which wildcards do not match them. Therefore, a pattern '`foo*bar`' can match a file name '`foo3/bar`', and a pattern '`./sr*sc`' can match a file name '`./src/misc`'.

If you want to locate some files with the '`locate`' command but don't need to see the full list you can use the '`--limit`' option to see just a small number of results, or the '`--count`' option to display only the total number of matches.

2.2 Links

There are two ways that files can be linked together. *Symbolic links* are a special type of file whose contents are a portion of the name of another file. *Hard links* are multiple directory entries for one file; the file names all have the same index node (*inode*) number on the disk.

2.2.1 Symbolic Links

Symbolic links are names that reference other files. GNU `find` will handle symbolic links in one of two ways; firstly, it can dereference the links for you - this means that if it comes across a symbolic link, it examines the file that the link points to, in order to see if it matches the criteria you have specified. Secondly, it can check the link itself in case you might be looking for the actual link. If the file that the symbolic link points to is also within the directory hierarchy you are searching with the `find` command, you may not see a great deal of difference between these two alternatives.

By default, `find` examines symbolic links themselves when it finds them (and, if it later comes across the linked-to file, it will examine that, too). If you would prefer `find` to dereference the links and examine the file that each link points to, specify the '`-L`' option to `find`. You can explicitly specify the default behaviour by using the '`-P`' option. The '`-H`' option is a half-way-between option which ensures that any symbolic links listed on the command line are dereferenced, but other symbolic links are not.

Symbolic links are different to "hard links" in the sense that you need permissions upon the linked-to file in order to be able to dereference the link. This can mean that even if you specify the '`-L`' option, find may not be able to determine the properties of the file that the link points to (because you don't have sufficient permissions). In this situation, '`find`' uses the properties of the link itself. This also occurs if a symbolic link exists but points to a file that is missing.

The options controlling the behaviour of `find` with respect to links are as follows :-

'`-P`' `find` does not dereference symbolic links at all. This is the default behaviour. This option must be specified before any of the path names on the command line.

'`-H`' `find` does not dereference symbolic links (except in the case of file names on the command line, which are dereferenced). If a symbolic link cannot be dereferenced, the information for the symbolic link itself is used. This option must be specified before any of the path names on the command line.

'`-L`' `find` dereferences symbolic links where possible, and where this is not possible it uses the properties of the symbolic link itself. This option must be specified before any of the path names on the command line. Use of this option also implies the same behaviour as the '`-noleaf`' option. If you later use the '`-H`' or '`-P`' options, this does not turn off '`-noleaf`'.

‘`-follow`’ This option forms part of the “expression” and must be specified after the path names, but it is otherwise equivalent to ‘`-L`’.

The following differences in behavior occur when the ‘`-L`’ option is used:

- `find` follows symbolic links to directories when searching directory trees.
- ‘`-lname`’ and ‘`-ilname`’ always return false (unless they happen to match broken symbolic links).
- ‘`-type`’ reports the types of the files that symbolic links point to.
- Implies ‘`-noleaf`’ (see Section 2.9 [Directories], page 12).

If the ‘`-L`’ option or the ‘`-H`’ option is used, the filenames used as arguments to ‘`-newer`’, ‘`-anewer`’, and ‘`-cnewer`’ are dereferenced and the timestamp from the pointed-to file is used instead (if possible – otherwise the timestamp from the symbolic link is used).

`-lname` *pattern* [Test]
`-ilname` *pattern* [Test]

True if the file is a symbolic link whose contents match shell pattern *pattern*. For ‘`-ilname`’, the match is case-insensitive. See Section 2.1.4 [Shell Pattern Matching], page 5, for details about the *pattern* argument. If the ‘`-L`’ option is in effect, this test will always fail for symbolic links unless they are broken. So, to list any symbolic links to ‘`sysdep.c`’ in the current directory and its subdirectories, you can do:

```
find . -lname '*sysdep.c'
```

2.2.2 Hard Links

Hard links allow more than one name to refer to the same file. To find all the names which refer to the same file as NAME, use ‘`-samefile NAME`’. If you are not using the ‘`-L`’ option, you can confine your search to one filesystem using the ‘`-xdev`’ option. This is useful because hard links cannot point outside a single filesystem, so this can cut down on needless searching.

If the ‘`-L`’ option is in effect, and NAME is in fact a symbolic link, the symbolic link will be dereferenced. Hence you are searching for other links (hard or symbolic) to the file pointed to by NAME. If ‘`-L`’ is in effect but NAME is not itself a symbolic link, other symbolic links to the file NAME will be matched.

You can also search for files by inode number. This can occasionally be useful in diagnosing problems with filesystems for example, because `fsck` tends to print inode numbers. Inode numbers also occasionally turn up in log messages for some types of software, and are used to support the `ftok()` library function.

You can learn a file’s inode number and the number of links to it by running ‘`ls -li`’ or ‘`find -ls`’.

You can search for hard links to inode number NUM by using ‘`-inum NUM`’. If there are any file system mount points below the directory where you are starting the search, use the ‘`-xdev`’ option unless you are also using the ‘`-L`’ option. Using ‘`-xdev`’ this saves needless searching, since hard links to a file must be on the same filesystem. See Section 2.10 [Filesystems], page 13.

`-samefile` *NAME* [Test]

File is a hard link to the same inode as NAME. If the ‘`-L`’ option is in effect, symbolic links to the same file as NAME points to are also matched.

`-inum` *n* [Test]

File has inode number *n*. The ‘`+`’ and ‘`-`’ qualifiers also work, though these are rarely useful.

You can also search for files that have a certain number of links, with ‘`-links`’. Directories normally have at least two hard links; their ‘`.`’ entry is the second one. If they have subdirectories, each of those also has a hard link called ‘`..`’ to its parent directory. The ‘`.`’ and ‘`..`’

directory entries are not normally searched unless they are mentioned on the `find` command line.

`-links` *n* [Test]
File has *n* hard links.

`-links +`*n* [Test]
File has more than *n* hard links.

`-links -`*n* [Test]
File has fewer than *n* hard links.

2.3 Time

Each file has three time stamps, which record the last time that certain operations were performed on the file:

1. access (read the file's contents)
2. change the status (modify the file or its attributes)
3. modify (change the file's contents)

There is no timestamp that indicates when a file was *created*.

You can search for files whose time stamps are within a certain age range, or compare them to other time stamps.

2.3.1 Age Ranges

These tests are mainly useful with ranges ('`+n`' and '`-n`').

`-atime` *n* [Test]
`-ctime` *n* [Test]
`-mtime` *n* [Test]
True if the file was last accessed (or its status changed, or it was modified) *n**24 hours ago. The number of 24-hour periods since the file's timestamp is always rounded down; therefore 0 means "less than 24 hours ago", 1 means "between 24 and 48 hours ago", and so forth.

`-amin` *n* [Test]
`-cmin` *n* [Test]
`-mmin` *n* [Test]
True if the file was last accessed (or its status changed, or it was modified) *n* minutes ago. These tests provide finer granularity of measurement than '`-atime`' et al., but rounding is done in a similar way. For example, to list files in '`/u/bill`' that were last read from 2 to 6 minutes ago:

```
find /u/bill -amin +2 -amin -6
```

`-daystart` [Option]
Measure times from the beginning of today rather than from 24 hours ago. So, to list the regular files in your home directory that were modified yesterday, do

```
find ~ -daystart -type f -mtime 1
```

The '`-daystart`' option is unlike most other options in that it has an effect on the way that other tests are performed. The affected tests are '`-amin`', '`-cmin`', '`-mmin`', '`-atime`', '`-ctime`' and '`-mtime`'.

2.3.2 Comparing Timestamps

As an alternative to comparing timestamps to the current time, you can compare them to another file's timestamp. That file's timestamp could be updated by another program when some event occurs. Or you could set it to a particular fixed date using the `touch` command. For example, to list files in '`/usr`' modified after February 1 of the current year:

```
touch -t 02010000 /tmp/stamp$$
find /usr -newer /tmp/stamp$$
rm -f /tmp/stamp$$
```

`-anewer` *file* [Test]
`-cnewer` *file* [Test]
`-newer` *file* [Test]

True if the file was last accessed (or its status changed, or it was modified) more recently than *file* was modified. These tests are affected by '`-follow`' only if '`-follow`' comes before them on the command line. See Section 2.2.1 [Symbolic Links], page 6, for more information on '`-follow`'. As an example, to list any files modified since '`/bin/sh`' was last modified:

```
find . -newer /bin/sh
```

`-used` *n* [Test]

True if the file was last accessed *n* days after its status was last changed. Useful for finding files that are not being used, and could perhaps be archived or removed to save disk space.

2.4 Size

`-size` *n*[*bckwMG*] [Test]

True if the file uses *n* units of space, rounding up. The units are 512-byte blocks by default, but they can be changed by adding a one-character suffix to *n*:

`b`	512-byte blocks (never 1024)
`c`	bytes
`k`	kilobytes (1024 bytes)
`w`	2-byte words
`M`	Megabytes
`G`	Gigabytes

The 'b' suffix always considers blocks to be 512 bytes. This is not affected by the setting (or non-setting) of the POSIXLY_CORRECT environment variable. This behaviour is different to the behaviour of the '`-ls`' action). If you want to use 1024-byte units, use the 'k' suffix instead.

The number can be prefixed with a '+' or a '-'. A plus sign indicates that the test should succeed if the file uses at least *n* units of storage (this is the way I normally use this test) and a minus sign indicates that the test should succeed if the file uses less than *n* units of storage. There is no '=' prefix, because that's the default anyway.

The size does not count indirect blocks, but it does count blocks in sparse files that are not actually allocated. In other words, it's consistent with the result you get for '`ls -l`' or '`wc -c`'. This handling of sparse files differs from the output of the '`%k`' and '`%b`' format specifiers for the '`-printf`' predicate.

`-empty` [Test]

True if the file is empty and is either a regular file or a directory. This might make it a good candidate for deletion. This test is useful with '`-depth`' (see Section 2.9 [Directories], page 12) and '`-delete`' (see Section 3.3.1 [Single File], page 19).

2.5 Type

`-type` *c* [Test]

True if the file is of type *c*:

`b`	block (buffered) special
`c`	character (unbuffered) special
`d`	directory
`p`	named pipe (FIFO)
`f`	regular file
`l`	symbolic link
`s`	socket
`D`	door (Solaris)

`-xtype` *c* [Test]

The same as '`-type`' unless the file is a symbolic link. For symbolic links: if '`-follow`' has not been given, true if the file is a link to a file of type *c*; if '`-follow`' has been given, true if *c* is '`l`'. In other words, for symbolic links, '`-xtype`' checks the type of the file that '`-type`' does not check. See Section 2.2.1 [Symbolic Links], page 6, for more information on '`-follow`'.

2.6 Owner

`-user` *uname* [Test]
`-group` *gname* [Test]

True if the file is owned by user *uname* (belongs to group *gname*). A numeric ID is allowed.

`-uid` *n* [Test]
`-gid` *n* [Test]

True if the file's numeric user ID (group ID) is *n*. These tests support ranges ('`+`*n*' and '`-`*n*'), unlike '`-user`' and '`-group`'.

`-nouser` [Test]
`-nogroup` [Test]

True if no user corresponds to the file's numeric user ID (no group corresponds to the numeric group ID). These cases usually mean that the files belonged to users who have since been removed from the system. You probably should change the ownership of such files to an existing user or group, using the `chown` or `chgrp` program.

2.7 Permissions

See Chapter 6 [File Permissions], page 31, for information on how file permissions are structured and how to specify them.

`-perm` *mode* [Test]

True if the file's permissions are exactly *mode* (which can be numeric or symbolic).

If *mode* starts with '`-`', true if *all* of the permissions set in *mode* are set for the file; permissions not set in *mode* are ignored. If *mode* starts with '`+`', true if *any* of the permissions set in *mode* are set for the file; permissions not set in *mode* are ignored.

If you don't use the '`+`' or '`-`' form with a symbolic mode string, you may have to specify a rather complex mode string. For example '`-perm g=w`' will only match files which have mode 0020 (that is, ones for which group write permission is the only permission set). It is more likely that you will want to use the '`+`' or '`-`' forms, for example '`-perm -g=w`', which matches any file with group write permission.

'`-perm 664`'
: Match files which have read and write permission for their owner, and group, but which the rest of the world can read but not write to. Files which meet these criteria but have other permissions bits set (for example if someone can execute the file) will not be matched.

'`-perm -664`'
: Match files which have read and write permission for their owner, and group, but which the rest of the world can read but not write to, without regard to the presence of any extra permission bits (for example the executable bit). This will match a file which has mode 0777, for example.

'`-perm +222`'
: Match files which are writeable by somebody (their owner, or their group, or anybody else).

'`-perm +022`'
: Match files which are writeable by either their owner or their group. The files don't have to be writeable by both the owner and group to be matched; either will do.

'`-perm +g+w,o+w`'
: As above.

'`-perm +g=w,o=w`'
: As above

'`-perm -022`'
: Search for files which are writeable by both their owner and their group.

'`-perm -g+w,o+w`'
: As above.

2.8 Contents

To search for files based on their contents, you can use the `grep` program. For example, to find out which C source files in the current directory contain the string '`thing`', you can do:

```
grep -l thing *.[ch]
```

If you also want to search for the string in files in subdirectories, you can combine `grep` with `find` and `xargs`, like this:

```
find . -name '*.[ch]' | xargs grep -l thing
```

The '`-l`' option causes `grep` to print only the names of files that contain the string, rather than the lines that contain it. The string argument ('`thing`') is actually a regular expression, so it can contain metacharacters. This method can be refined a little by using the '`-r`' option to make `xargs` not run `grep` if `find` produces no output, and using the `find` action '`-print0`' and the `xargs` option '`-0`' to avoid misinterpreting files whose names contain spaces:

```
find . -name '*.[ch]' -print0 | xargs -r -0 grep -l thing
```

For a fuller treatment of finding files whose contents match a pattern, see the manual page for `grep`.

2.9 Directories

Here is how to control which directories `find` searches, and how it searches them. These two options allow you to process a horizontal slice of a directory tree.

`-maxdepth` *levels* [Option]

Descend at most *levels* (a non-negative integer) levels of directories below the command line arguments. '`-maxdepth 0`' means only apply the tests and actions to the command line arguments.

`-mindepth` *levels* [Option]

Do not apply any tests or actions at levels less than *levels* (a non-negative integer). '`-mindepth 1`' means process all files except the command line arguments.

`-depth` [Option]

Process each directory's contents before the directory itself. Doing this is a good idea when producing lists of files to archive with `cpio` or `tar`. If a directory does not have write permission for its owner, its contents can still be restored from the archive since the directory's permissions are restored after its contents.

`-d` [Option]

This is a deprecated synonym for '`-depth`', for compatibility with Mac OS X, FreeBSD and OpenBSD. The '`-depth`' option is a POSIX feature, so it is better to use that.

`-prune` [Action]

If the file is a directory, do not descend into it. The result is true. For example, to skip the directory '`src/emacs`' and all files and directories under it, and print the names of the other files found:

```
find . -wholename './src/emacs' -prune -o -print
```

The above command will not print '`./src/emacs`' among its list of results. This however is not due to the effect of the '`-prune`' action (which only prevents further descent, it doesn't make sure we ignore that item). Instead, this effect is due to the use of '`-o`'. Since the left hand side of the "or" condition has succeeded for '`./src/emacs`', it is not necessary to evaluate the right-hand-side ('`-print`') at all for this particular file. If you wanted to print that directory name you could use either an extra '`-print`' action:

```
find . -wholename './src/emacs' -prune -print -o -print
```

or use the comma operator:

```
find . -wholename './src/emacs' -prune , -print
```

If the '`-depth`' option is in effect, the subdirectories will have already been visited in any case. Hence '`-prune`' has no effect and returns false.

`-quit` [Action]

Exit immediately (with return value zero if no errors have occurred). No child processes will be left running, but no more paths specified on the command line will be processed. For example, `find /tmp/foo /tmp/bar -print -quit` will print only '`/tmp/foo`'.

`-noleaf` [Option]

Do not optimize by assuming that directories contain 2 fewer subdirectories than their hard link count. This option is needed when searching filesystems that do not follow the Unix directory-link convention, such as CD-ROM or MS-DOS filesystems or AFS volume mount points. Each directory on a normal Unix filesystem has at least 2 hard links: its name and its '`.`' entry. Additionally, its subdirectories (if any) each have a '`..`' entry linked to that directory. When `find` is examining a directory, after it has statted 2 fewer subdirectories than the directory's link count, it knows that the rest of the entries in the directory are non-directories (*leaf* files in the directory tree). If only the files' names need to be examined, there is no need to stat them; this gives a significant increase in search speed.

`-ignore_readdir_race` [Option]

If a file disappears after its name has been read from a directory but before `find` gets around to examining the file with `stat`, don't issue an error message. If you don't specify this option, an error message will be issued. This option can be useful in system scripts (cron scripts, for example) that examine areas of the filesystem that change frequently (mail queues, temporary directories, and so forth), because this scenario is common for those sorts of directories. Completely silencing error messages from `find` is undesirable, so this option neatly solves the problem. There is no way to search one part of the filesystem with this option on and part of it with this option off, though.

`-noignore_readdir_race` [Option]

This option reverses the effect of the '`-ignore_readdir_race`' option.

2.10 Filesystems

A *filesystem* is a section of a disk, either on the local host or mounted from a remote host over a network. Searching network filesystems can be slow, so it is common to make `find` avoid them.

There are two ways to avoid searching certain filesystems. One way is to tell `find` to only search one filesystem:

`-xdev` [Option]
`-mount` [Option]

Don't descend directories on other filesystems. These options are synonyms.

The other way is to check the type of filesystem each file is on, and not descend directories that are on undesirable filesystem types:

`-fstype` *type* [Test]

True if the file is on a filesystem of type *type*. The valid filesystem types vary among different versions of Unix; an incomplete list of filesystem types that are accepted on some version of Unix or another is:

```
ext2 ext3 proc sysfs ufs 4.2 4.3 nfs tmp mfs S51K S52K
```

You can use '`-printf`' with the '`%F`' directive to see the types of your filesystems. The '`%D`' directive shows the device number. See Section 3.2 [Print File Information], page 15. '`-fstype`' is usually used with '`-prune`' to avoid searching remote filesystems (see Section 2.9 [Directories], page 12).

2.11 Combining Primaries With Operators

Operators build a complex expression from tests and actions. The operators are, in order of decreasing precedence:

`(` *expr* `)` Force precedence. True if *expr* is true.

`!` *expr*
`-not` *expr*

True if *expr* is false.

expr1 expr2
expr1 `-a` *expr2*
expr1 `-and` *expr2*

And; *expr2* is not evaluated if *expr1* is false.

`*expr1* -o *expr2*`
`*expr1* -or *expr2*`
: Or; *expr2* is not evaluated if *expr1* is true.

`*expr1* , *expr2*`
: List; both *expr1* and *expr2* are always evaluated. True if *expr2* is true. The value of *expr1* is discarded. This operator lets you do multiple independent operations on one traversal, without depending on whether other operations succeeded. The two operations *expr1* and *expr2* are not always fully independent, since *expr1* might have side effects like touching or deleting files, or it might use '`-prune`' which would also affect *expr2*.

`find` searches the directory tree rooted at each file name by evaluating the expression from left to right, according to the rules of precedence, until the outcome is known (the left hand side is false for '`-and`', true for '`-or`'), at which point `find` moves on to the next file name.

There are two other tests that can be useful in complex expressions:

`-true` [Test]
Always true.

`-false` [Test]
Always false.

3 Actions

There are several ways you can print information about the files that match the criteria you gave in the `find` expression. You can print the information either to the standard output or to a file that you name. You can also execute commands that have the file names as arguments. You can use those commands as further filters to select files.

3.1 Print File Name

`-print` [Action]
True; print the full file name on the standard output, followed by a newline.

`-fprint` *file* [Action]
True; print the full file name into file *file*, followed by a newline. If *file* does not exist when `find` is run, it is created; if it does exist, it is truncated to 0 bytes. The file names '`/dev/stdout`' and '`/dev/stderr`' are handled specially; they refer to the standard output and standard error output, respectively.

3.2 Print File Information

`-ls` [Action]
True; list the current file in '`ls -dils`' format on the standard output. The output looks like this:

```
      204744   17 -rw-r--r--   1 djm      staff       17337 Nov  2  1992 ./lwall-quotes
```

The fields are:

1. The inode number of the file. See Section 2.2.2 [Hard Links], page 7, for how to find files based on their inode number.
2. the number of blocks in the file. The block counts are of 1K blocks, unless the environment variable `POSIXLY_CORRECT` is set, in which case 512-byte blocks are used. See Section 2.4 [Size], page 9, for how to find files based on their size.
3. The file's type and permissions. The type is shown as a dash for a regular file; for other file types, a letter like for '`-type`' is used (see Section 2.5 [Type], page 10). The permissions are read, write, and execute for the file's owner, its group, and other users, respectively; a dash means the permission is not granted. See Chapter 6 [File Permissions], page 31, for more details about file permissions. See Section 2.7 [Permissions], page 10, for how to find files based on their permissions.
4. The number of hard links to the file.
5. The user who owns the file.
6. The file's group.
7. The file's size in bytes.
8. The date the file was last modified.
9. The file's name. '`-ls`' quotes non-printable characters in the file names using C-like backslash escapes.

`-fls` *file* [Action]
True; like '`-ls`' but write to *file* like '`-fprint`' (see Section 3.1 [Print File Name], page 15).

`-printf` *format* [Action]
True; print *format* on the standard output, interpreting '\' escapes and '`%`' directives. Field widths and precisions can be specified as with the `printf` C function. Format flags (like '`#`'

for example) may not work as you expect because many of the fields, even numeric ones, are printed with %s. This means though that the format flag '`-`' will work; it forces left-alignment of the field. Unlike '`-print`', '`-printf`' does not add a newline at the end of the string. If you want a newline at the end of the string, add a '`\n`'.

`-fprintf` *file format* [Action]
True; like '`-printf`' but write to *file* like '`-fprint`' (see Section 3.1 [Print File Name], page 15).

3.2.1 Escapes

The escapes that '`-printf`' and '`-fprintf`' recognize are:

`\a`	Alarm bell.
`\b`	Backspace.
`\c`	Stop printing from this format immediately and flush the output.
`\f`	Form feed.
`\n`	Newline.
`\r`	Carriage return.
`\t`	Horizontal tab.
`\v`	Vertical tab.
`\\`	A literal backslash ('\').
`\NNN`	The character whose ASCII code is NNN (octal).

A '\' character followed by any other character is treated as an ordinary character, so they both are printed, and a warning message is printed to the standard error output (because it was probably a typo).

3.2.2 Format Directives

'`-printf`' and '`-fprintf`' support the following format directives to print information about the file being processed. The C `printf` function, field width and precision specifiers are supported, as applied to string (%s) types. That is, you can specify "minimum field width"."maximum field width" for each directive. Format flags (like '`#`' for example) may not work as you expect because many of the fields, even numeric ones, are printed with %s. The format flag '`-`' does work; it forces left-alignment of the field.

'`%%`' is a literal percent sign. A '`%`' character followed by an unrecognised character (i.e. not a known directive or printf field width and precision specifier), is discarded (but the unrecognised character is printed), and a warning message is printed to the standard error output (because it was probably a typo).

3.2.2.1 Name Directives

`%p`	File's name (not the absolute path name, but the name of the file as it was encountered by find - that is, as a relative path from one of the starting points).
`%f`	File's name with any leading directories removed (only the last element).
`%h`	Leading directories of file's name (all but the last element and the slash before it). If the file's name contains no slashes (for example because it was named on the command line and is in the current working directory), then "%h" expands to ".". This prevents "%h/%f" expanding to "/foo", which would be surprising and probably not desirable.

`%P` File's name with the name of the command line argument under which it was found removed from the beginning.

`%H` Command line argument under which file was found.

3.2.2.2 Ownership Directives

`%g` File's group name, or numeric group ID if the group has no name.

`%G` File's numeric group ID.

`%u` File's user name, or numeric user ID if the user has no name.

`%U` File's numeric user ID.

`%m` File's permissions (in octal). If you always want to have a leading zero on the number, use the '#' format flag, for example '%#m'.

3.2.2.3 Size Directives

`%k` Amount of disk space occupied by the file, measured in 1K blocks (rounded up). This can be less than the length of the file if it is a sparse file (that is, it has "holes").

`%b` File's size in 512-byte blocks (rounded up). This also can be less than the length of the file, if the file is sparse.

`%s` File's size in bytes.

3.2.2.4 Location Directives

`%d` File's depth in the directory tree (depth below a file named on the command line, not depth below the root directory). Files named on the command line have a depth of 0. Subdirectories immediately below them have a depth of 1, and so on.

`%D` The device number on which the file exists (the `st_dev` field of `struct stat`), in decimal.

`%F` Type of the filesystem the file is on; this value can be used for '`-fstype`' (see Section 2.9 [Directories], page 12).

`%l` Object of symbolic link (empty string if file is not a symbolic link).

`%i` File's inode number (in decimal).

`%n` Number of hard links to file.

`%y` Type of the file as used with '`-type`'. If the file is a symbolic link, '`l`' will be printed.

`%Y` Type of the file as used with '`-type`'. If the file is a symbolic link, it is dereferenced. If the file is a broken symbolic link, '`N`' is printed.

3.2.2.5 Time Directives

Some of these directives use the C `ctime` function. Its output depends on the current locale, but it typically looks like

```
Wed Nov  2 00:42:36 1994
```

`%a` File's last access time in the format returned by the C `ctime` function.

`%A`*k* File's last access time in the format specified by *k* (see Section 3.2.3 [Time Formats], page 18).

`%c` File's last status change time in the format returned by the C `ctime` function.

`%C`*k*	File's last status change time in the format specified by *k* (see Section 3.2.3 [Time Formats], page 18).
`%t`	File's last modification time in the format returned by the C `ctime` function.
`%T`*k*	File's last modification time in the format specified by *k* (see Section 3.2.3 [Time Formats], page 18).

3.2.3 Time Formats

Below are the formats for the directives '`%A`', '`%C`', and '`%T`', which print the file's timestamps. Some of these formats might not be available on all systems, due to differences in the C `strftime` function between systems.

3.2.3.1 Time Components

The following format directives print single components of the time.

`H`	hour (00..23)
`I`	hour (01..12)
`k`	hour (0..23)
`l`	hour (1..12)
`p`	locale's AM or PM
`Z`	time zone (e.g., EDT), or nothing if no time zone is determinable
`M`	minute (00..59)
`S`	second (00..61)
`@`	seconds since Jan. 1, 1970, 00:00 GMT.

3.2.3.2 Date Components

The following format directives print single components of the date.

`a`	locale's abbreviated weekday name (Sun..Sat)
`A`	locale's full weekday name, variable length (Sunday..Saturday)
`b` `h`	locale's abbreviated month name (Jan..Dec)
`B`	locale's full month name, variable length (January..December)
`m`	month (01..12)
`d`	day of month (01..31)
`w`	day of week (0..6)
`j`	day of year (001..366)
`U`	week number of year with Sunday as first day of week (00..53)
`W`	week number of year with Monday as first day of week (00..53)
`Y`	year (1970. . .)
`y`	last two digits of year (00..99)

3.2.3.3 Combined Time Formats

The following format directives print combinations of time and date components.

`r`	time, 12-hour (hh:mm:ss [AP]M)
`T`	time, 24-hour (hh:mm:ss)
`X`	locale's time representation (H:M:S)
`c`	locale's date and time (Sat Nov 04 12:02:33 EST 1989)
`D`	date (mm/dd/yy)
`x`	locale's date representation (mm/dd/yy)
`+`	Date and time, separated by '+', for example '2004-04-28+22:22:05'. The time is given in the current timezone (which may be affected by setting the TZ environment variable). This is a GNU extension.

3.2.3.4 Formatting Flags

The '`%m`' and '`%d`' directives support the '`#`', '`0`' and '`+`' flags, but the other directives do not, even if they print numbers. Numeric directives that do not support these flags include

'`G`', '`U`', '`b`', '`D`', '`k`' and '`n`'.

All fields support the format flag '`-`', which makes fields left-aligned. That is, if the field width is greater than the actual contents of the field, the requisite number of spaces are printed after the field content instead of before it.

3.3 Run Commands

You can use the list of file names created by `find` or `locate` as arguments to other commands. In this way you can perform arbitrary actions on the files.

3.3.1 Single File

Here is how to run a command on one file at a time.

`-execdir` *command ;* [Action]

Execute *command*; true if 0 status is returned. `find` takes all arguments after '`-exec`' to be part of the command until an argument consisting of '`;`' is reached. It replaces the string '`{}`' by the current file name being processed everywhere it occurs in the command. Both of these constructions need to be escaped (with a '\') or quoted to protect them from expansion by the shell. The command is executed in the directory in which `find` was run.

For example, to compare each C header file in the current directory with the file '`/tmp/master`':

```
find . -name '*.h' -execdir diff -u '{}' /tmp/master ';'
```

Another similar option, '`-exec`' is supported, but is less secure. See Chapter 8 [Security Considerations], page 41, for a discussion of the security problems surrounding '`-exec`'.

`-exec` *command ;* [Action]

This insecure variant of the '`-execdir`' action is specified by POSIX. The main difference is that the command is executed in the directory from which `find` was invoked, meaning that '`{}`' is expanded to a relative path starting with the name of one of the starting directories, rather than just the basename of the matched file.

3.3.2 Multiple Files

Sometimes you need to process files one of the time. But usually this is not necessary, and, it is faster to run a command on as many files as possible at a time, rather than once per file. Doing this saves on the time it takes to start up the command each time.

The '`-execdir`' and '`-exec`' actions have variants that build command lines containing as many matched files as possible.

`-execdir` *command* `{} +` [Action]

This works as for '`-execdir command ;`', except that the '`{}`' at the end of the command is expanded to a list of names of matching files. This expansion is done in such a way as to avoid exceeding the maximum command line length available on the system. Only one '`{}`' is allowed within the command, and it must appear at the end, immediately before the '`+`'. A '`+`' appearing in any position other than immediately after '`{}`' is not considered to be special (that is, it does not terminate the command).

`-exec` *command* `{} +` [Action]

This insecure variant of the '`-execdir`' action is specified by POSIX. The main difference is that the command is executed in the directory from which `find` was invoked, meaning that '`{}`' is expanded to a relative path starting with the name of one of the starting directories, rather than just the basename of the matched file.

Before `find` exits, any partially-built command lines are executed. This happens even if the exit was caused by the '`-quit`' action. However, some types of error (for example not being able to invoke `stat()` on the current directory) can cause an immediate fatal exit. In this situation, any partially-built command lines will not be invoked (this prevents possible infinite loops).

Another, but less secure, way to run a command on more than one file at once, is to use the `xargs` command, which is invoked like this:

```
xargs [option ...] [command [initial-arguments]]
```

`xargs` normally reads arguments from the standard input. These arguments are delimited by blanks (which can be protected with double or single quotes or a backslash) or newlines. It executes the *command* (default is '`/bin/echo`') one or more times with any *initial-arguments* followed by arguments read from standard input. Blank lines on the standard input are ignored.

Instead of blank-delimited names, it is safer to use '`find -print0`' or '`find -fprint0`' and process the output by giving the '`-0`' or '`--null`' option to GNU `xargs`, GNU `tar`, GNU `cpio`, or `perl`. The `locate` command also has a '`-0`' or '`--null`' option which does the same thing.

You can use shell command substitution (backquotes) to process a list of arguments, like this:

```
grep -l sprintf ‘find $HOME -name ’*.c’ -print‘
```

However, that method produces an error if the length of the '`.c`' file names exceeds the operating system's command-line length limit. `xargs` avoids that problem by running the command as many times as necessary without exceeding the limit:

```
find $HOME -name ’*.c’ -print | xargs grep -l sprintf
```

However, if the command needs to have its standard input be a terminal (`less`, for example), you have to use the shell command substitution method or use the '`--arg-file`' option of `xargs`.

The `xargs` command will process all its input, building command lines and executing them, unless one of the commands exits with a status of 255 (this will cause xargs to issue an error message and stop) or it reads a line contains the end of file string specified with the '`--eof`' option.

3.3.2.1 Unsafe File Name Handling

Because file names can contain quotes, backslashes, blank characters, and even newlines, it is not safe to process them using `xargs` in its default mode of operation. But since most files' names do not contain blanks, this problem occurs only infrequently. If you are only searching through files that you know have safe names, then you need not be concerned about it.

In many applications, if `xargs` botches processing a file because its name contains special characters, some data might be lost. The importance of this problem depends on the importance of the data and whether anyone notices the loss soon enough to correct it. However, here is an extreme example of the problems that using blank-delimited names can cause. If the following command is run daily from `cron`, then any user can remove any file on the system:

```
find / -name '#*' -atime +7 -print | xargs rm
```

For example, you could do something like this:

```
eg$ echo > '#
vmunix'
```

and then `cron` would delete '`/vmunix`', if it ran `xargs` with '`/`' as its current directory.

To delete other files, for example '`/u/joeuser/.plan`', you could do this:

```
eg$ mkdir '#
'
eg$ cd '#
'
eg$ mkdir u u/joeuser u/joeuser/.plan'
'
eg$ echo > u/joeuser/.plan'
/#foo'
eg$ cd ..
eg$ find . -name '#*' -print | xargs echo
./# ./# /u/joeuser/.plan /#foo
```

3.3.2.2 Safe File Name Handling

Here is how to make `find` output file names so that they can be used by other programs without being mangled or misinterpreted. You can process file names generated this way by giving the '`-0`' or '`--null`' option to GNU `xargs`, GNU `tar`, GNU `cpio`, or `perl`.

`-print0` [Action]
: True; print the full file name on the standard output, followed by a null character.

`-fprint0` *file* [Action]
: True; like '`-print0`' but write to *file* like '`-fprint`' (see Section 3.1 [Print File Name], page 15).

As of findutils version 4.2.4, the `locate` program also has a '`--null`' option which does the same thing. For similarity with `xargs`, the short form of the option '`-0`' can also be used.

If you want to be able to handle file names safely but need to run commands which want to be connected to a terminal on their input, you can use the '`--arg-file`' option to `xargs` like this:

```
find / -name xyzzy -print0 > list
xargs --null --arg-file=list munge
```

The example above runs the `munge` program on all the files named '`xyzzy`' that we can find, but `munge`'s input will still be the terminal (or whatever the shell was using as standard input). If your shell has the "process substitution" feature '`<(...)`', you can do this in just one step:

```
xargs --null --arg-file=<(find / -name xyzzy -print0) munge
```

3.3.2.3 Limiting Command Size

`xargs` gives you control over how many arguments it passes to the command each time it executes it. By default, it uses up to `ARG_MAX` - 2k, or 128k, whichever is smaller, characters per command. It uses as many lines and arguments as fit within that limit. The following options modify those values.

`--no-run-if-empty`
`-r` If the standard input does not contain any nonblanks, do not run the command. By default, the command is run once even if there is no input.

`--max-lines`[=*`max-lines`*]
`-l`[*`max-lines`*]
Use at most *max-lines* nonblank input lines per command line; *max-lines* defaults to 1 if omitted. Trailing blanks cause an input line to be logically continued on the next input line, for the purpose of counting the lines. Implies '`-x`'.

`--max-args=`*`max-args`*
`-n` *`max-args`*
Use at most *max-args* arguments per command line. Fewer than *max-args* arguments will be used if the size (see the '`-s`' option) is exceeded, unless the '`-x`' option is given, in which case `xargs` will exit.

`--max-chars=`*`max-chars`*
`-s` *`max-chars`*
Use at most *max-chars* characters per command line, including the command and initial arguments and the terminating nulls at the ends of the argument strings. If you specify a value for this option which is too large or small, a warning message is printed and the appropriate upper or lower limit is used instead.

`--max-procs=`*`max-procs`*
`-P` *`max-procs`*
Run up to *max-procs* processes at a time; the default is 1. If *max-procs* is 0, `xargs` will run as many processes as possible at a time. Use the '`-n`', '`-s`', or '`-l`' option with '`-P`'; otherwise chances are that the command will be run only once.

3.3.2.4 Interspersing File Names

`xargs` can insert the name of the file it is processing between arguments you give for the command. Unless you also give options to limit the command size (see Section 3.3.2.3 [Limiting Command Size], page 22), this mode of operation is equivalent to '`find -exec`' (see Section 3.3.1 [Single File], page 19).

`--replace`[=*`replace-str`*]
`-i`[*`replace-str`*]
Replace occurrences of *replace-str* in the initial arguments with names read from the input. Also, unquoted blanks do not terminate arguments; instead, the input is split at newlines only. If *replace-str* is omitted, it defaults to '`{}`' (like for '`find -exec`'). Implies '`-x`' and '`-l 1`'. As an example, to sort each file the '`bills`' directory, leaving the output in that file name with '`.sorted`' appended, you could do:

```
find bills -type f | xargs -iXX sort -o XX.sorted XX
```

The equivalent command using '`find -exec`' is:

```
find bills -type f -exec sort -o '{}.sorted' '{}' ';'
```

3.3.3 Querying

To ask the user whether to execute a command on a single file, you can use the `find` primary '`-ok`' instead of '`-exec`':

`-ok` *command ;* [Action]

Like '`-exec`' (see Section 3.3.1 [Single File], page 19), but ask the user first (on the standard input); if the response does not start with '`y`' or '`Y`', do not run the command, and return false.

When processing multiple files with a single command, to query the user you give `xargs` the following option. When using this option, you might find it useful to control the number of files processed per invocation of the command (see Section 3.3.2.3 [Limiting Command Size], page 22).

`--interactive`
`-p` Prompt the user about whether to run each command line and read a line from the terminal. Only run the command line if the response starts with '`y`' or '`Y`'. Implies '`-t`'.

3.4 Delete Files

`-delete` [Action]

Delete files or directories; true if removal succeeded. If the removal failed, an error message is issued.

The use of the '`-delete`' action on the command line automatically turns on the '`-depth`' option (see Section 1.3 [find Expressions], page 2).

3.5 Adding Tests

You can test for file attributes that none of the `find` builtin tests check. To do this, use `xargs` to run a program that filters a list of files printed by `find`. If possible, use `find` builtin tests to pare down the list, so the program run by `xargs` has less work to do. The tests builtin to `find` will likely run faster than tests that other programs perform.

For reasons of efficiency it is often useful to limit the number of times an external program has to be run. For this reason, it is often a good idea to implement "extended" tests by using `xargs`.

For example, here is a way to print the names of all of the unstripped binaries in the '`/usr/local`' directory tree. Builtin tests avoid running `file` on files that are not regular files or are not executable.

```
find /usr/local -type f -perm +a=x | xargs file |
  grep 'not stripped' | cut -d: -f1
```

The `cut` program removes everything after the file name from the output of `file`.

However, using `xargs` can present important security problems (see Chapter 8 [Security Considerations], page 41). These can be avoided by using '`-execdir`'. The '`-execdir`' action is also a useful way of putting your own test in the middle of a set of other tests or actions for `find` (for example, you might want to use '`-prune`').

To place a special test somewhere in the middle of a `find` expression, you can use '`-execdir`' (or, less securely, '`-exec`') to run a program that performs the test. Because '`-execdir`' evaluates to the exit status of the executed program, you can use a program (which can be a shell script) that tests for a special attribute and make it exit with a true (zero) or false (non-zero) status. It is a good idea to place such a special test *after* the builtin tests, because it starts a new process which could be avoided if a builtin test evaluates to false.

Here is a shell script called `unstripped` that checks whether its argument is an unstripped binary file:

```
#! /bin/sh
file "$1" | grep -q "not stripped"
```

This script relies on the fact that the shell exits with the status of the last command in the pipeline, in this case `grep`. The `grep` command exits with a true status if it found any matches, false if not. Here is an example of using the script (assuming it is in your search path). It lists the stripped executables (and shell scripts) in the file '`sbins`' and the unstripped ones in '`ubins`'.

```
find /usr/local -type f -perm +a=x \
  \( -execdir unstripped '{}' \; -fprint ubins -o -fprint sbins \)
```

4 Common Tasks

The sections that follow contain some extended examples that both give a good idea of the power of these programs, and show you how to solve common real-world problems.

4.1 Viewing And Editing

To view a list of files that meet certain criteria, simply run your file viewing program with the file names as arguments. Shells substitute a command enclosed in backquotes with its output, so the whole command looks like this:

```
less `find /usr/include -name '*.h' | xargs grep -l mode_t`
```

You can edit those files by giving an editor name instead of a file viewing program:

```
emacs `find /usr/include -name '*.h' | xargs grep -l mode_t`
```

Because there is a limit to the length of any individual command line, there is a limit to the number of files that can be handled in this way. We can get around this difficulty by using xargs like this:

```
find /usr/include -name '*.h' | xargs grep -l mode_t > todo
xargs --arg-file=todo emacs
```

Here, `xargs` will run `emacs` as many times as necessary to visit all of the files listed in the file '`todo`'.

4.2 Archiving

You can pass a list of files produced by `find` to a file archiving program. GNU `tar` and `cpio` can both read lists of file names from the standard input—either delimited by nulls (the safe way) or by blanks (the lazy, risky default way). To use null-delimited names, give them the '`--null`' option. You can store a file archive in a file, write it on a tape, or send it over a network to extract on another machine.

One common use of `find` to archive files is to send a list of the files in a directory tree to `cpio`. Use '`-depth`' so if a directory does not have write permission for its owner, its contents can still be restored from the archive since the directory's permissions are restored after its contents. Here is an example of doing this using `cpio`; you could use a more complex `find` expression to archive only certain files.

```
find . -depth -print0 |
  cpio --create --null --format=crc --file=/dev/nrst0
```

You could restore that archive using this command:

```
cpio --extract --null --make-dir --unconditional \
  --preserve --file=/dev/nrst0
```

Here are the commands to do the same things using `tar`:

```
find . -depth -print0 |
  tar --create --null --files-from=- --file=/dev/nrst0

tar --extract --null --preserve-perm --same-owner \
  --file=/dev/nrst0
```

Here is an example of copying a directory from one machine to another:

```
find . -depth -print0 | cpio -0o -Hnewc |
  rsh other-machine "cd `pwd` && cpio -i0dum"
```

4.3 Cleaning Up

This section gives examples of removing unwanted files in various situations. Here is a command to remove the CVS backup files created when an update requires a merge:

```
find . -name '.#*' -print0 | xargs -0r rm -f
```

The command above works, but the following is safer:

```
find . -name '.#*' -depth -delete
```

You can run this command to clean out your clutter in '`/tmp`'. You might place it in the file your shell runs when you log out ('`.bash_logout`', '`.logout`', or '`.zlogout`', depending on which shell you use).

```
find /tmp -depth -user "$LOGNAME" -type f -delete
```

If your `find` command removes directories, you may find that you get a spurious error message when `find` tries to recurse into a directory that has now been removed. Using the '`-depth`' option will normally resolve this problem.

To remove old Emacs backup and auto-save files, you can use a command like the following. It is especially important in this case to use null-terminated file names because Emacs packages like the VM mailer often create temporary file names with spaces in them, like '`#reply to David J. MacKenzie<1>#`'.

```
find ~ \( -name '*~' -o -name '#*#' \) -print0 |
  xargs --no-run-if-empty --null rm -vf
```

Removing old files from '`/tmp`' is commonly done from `cron`:

```
find /tmp /var/tmp -not -type d -mtime +3 -delete
find /tmp /var/tmp -depth -mindepth 1 -type d -empty -delete
```

The second `find` command above uses '`-depth`' so it cleans out empty directories depth-first, hoping that the parents become empty and can be removed too. It uses '`-mindepth`' to avoid removing '`/tmp`' itself if it becomes totally empty.

4.4 Strange File Names

`find` can help you remove or rename a file with strange characters in its name. People are sometimes stymied by files whose names contain characters such as spaces, tabs, control characters, or characters with the high bit set. The simplest way to remove such files is:

```
rm -i some*pattern*that*matches*the*problem*file
```

`rm` asks you whether to remove each file matching the given pattern. If you are using an old shell, this approach might not work if the file name contains a character with the high bit set; the shell may strip it off. A more reliable way is:

```
find . -maxdepth 1 tests -ok rm '{}' \;
```

where *tests* uniquely identify the file. The '`-maxdepth 1`' option prevents `find` from wasting time searching for the file in any subdirectories; if there are no subdirectories, you may omit it. A good way to uniquely identify the problem file is to figure out its inode number; use

```
ls -i
```

Suppose you have a file whose name contains control characters, and you have found that its inode number is 12345. This command prompts you for whether to remove it:

```
find . -maxdepth 1 -inum 12345 -ok rm -f '{}' \;
```

If you don't want to be asked, perhaps because the file name may contain a strange character sequence that will mess up your screen when printed, then use '`-exec`' instead of '`-ok`'.

If you want to rename the file instead, you can use `mv` instead of `rm`:

```
find . -maxdepth 1 -inum 12345 -ok mv '{}' new-file-name \;
```

4.5 Fixing Permissions

Suppose you want to make sure that everyone can write to the directories in a certain directory tree. Here is a way to find directories lacking either user or group write permission (or both), and fix their permissions:

```
find . -type d -not -perm -ug=w | xargs chmod ug+w
```

You could also reverse the operations, if you want to make sure that directories do *not* have world write permission.

4.6 Classifying Files

If you want to classify a set of files into several groups based on different criteria, you can use the comma operator to perform multiple independent tests on the files. Here is an example:

```
find / -type d \( -perm -o=w -fprint allwrite , \
  -perm -o=x -fprint allexec \)

echo "Directories that can be written to by everyone:"
cat allwrite
echo ""
echo "Directories with search permissions for everyone:"
cat allexec
```

`find` has only to make one scan through the directory tree (which is one of the most time consuming parts of its work).

5 File Name Databases

The file name databases used by `locate` contain lists of files that were in particular directory trees when the databases were last updated. The file name of the default database is determined when `locate` and `updatedb` are configured and installed. The frequency with which the databases are updated and the directories for which they contain entries depend on how often `updatedb` is run, and with which arguments.

You can obtain some statistics about the databases by using '`locate --statistics`'.

5.1 Database Locations

There can be multiple file name databases. Users can select which databases `locate` searches using the `LOCATE_PATH` environment variable or a command line option. The system administrator can choose the file name of the default database, the frequency with which the databases are updated, and the directories for which they contain entries. File name databases are updated by running the `updatedb` program, typically nightly.

In networked environments, it often makes sense to build a database at the root of each filesystem, containing the entries for that filesystem. `updatedb` is then run for each filesystem on the fileserver where that filesystem is on a local disk, to prevent thrashing the network.

See Section 7.3 [Invoking updatedb], page 38, for the description of the options to `updatedb`, which specify which directories would each database contain entries for.

5.2 Database Formats

The file name databases contain lists of files that were in particular directory trees when the databases were last updated. The file name database format changed starting with GNU `locate` version 4.0 to allow machines with different byte orderings to share the databases. The new GNU `locate` can read both the old and new database formats. However, old versions of `locate` and `find` produce incorrect results if given a new-format database.

If you run '`locate --statistics`', the resulting summary indicates the type of each locate database.

5.2.1 New Database Format

`updatedb` runs a program called `frcode` to *front-compress* the list of file names, which reduces the database size by a factor of 4 to 5. Front-compression (also known as incremental encoding) works as follows.

The database entries are a sorted list (case-insensitively, for users' convenience). Since the list is sorted, each entry is likely to share a prefix (initial string) with the previous entry. Each database entry begins with an offset-differential count byte, which is the additional number of characters of prefix of the preceding entry to use beyond the number that the preceding entry is using of its predecessor. (The counts can be negative.) Following the count is a null-terminated ASCII remainder—the part of the name that follows the shared prefix.

If the offset-differential count is larger than can be stored in a byte (+/-127), the byte has the value 0x80 and the count follows in a 2-byte word, with the high byte first (network byte order).

Every database begins with a dummy entry for a file called '`LOCATE02`', which `locate` checks for to ensure that the database file has the correct format; it ignores the entry in doing the search.

Databases can not be concatenated together, even if the first (dummy) entry is trimmed from all but the first database. This is because the offset-differential count in the first entry of the second and following databases will be wrong.

In the output of '`locate --statistics`', the new database format is referred to as '`LOCATE02`'.

5.2.2 Sample Database

Sample input to `frcode`:

```
/usr/src
/usr/src/cmd/aardvark.c
/usr/src/cmd/armadillo.c
/usr/tmp/zoo
```

Length of the longest prefix of the preceding entry to share:

```
0 /usr/src
8 /cmd/aardvark.c
14 rmadillo.c
5 tmp/zoo
```

Output from `frcode`, with trailing nulls changed to newlines and count bytes made printable:

```
0 LOCATE02
0 /usr/src
8 /cmd/aardvark.c
6 rmadillo.c
-9 tmp/zoo
```

(6 = 14 - 8, and -9 = 5 - 14)

5.2.3 Old Database Format

The old database format is used by Unix `locate` and `find` programs and earlier releases of the GNU ones. `updatedb` produces this format if given the '`--old-format`' option.

`updatedb` runs programs called `bigram` and `code` to produce old-format databases. The old format differs from the new one in the following ways. Instead of each entry starting with an offset-differential count byte and ending with a null, byte values from 0 through 28 indicate offset-differential counts from -14 through 14. The byte value indicating that a long offset-differential count follows is 0x1e (30), not 0x80. The long counts are stored in host byte order, which is not necessarily network byte order, and host integer word size, which is usually 4 bytes. They also represent a count 14 less than their value. The database lines have no termination byte; the start of the next line is indicated by its first byte having a value <= 30.

In addition, instead of starting with a dummy entry, the old database format starts with a 256 byte table containing the 128 most common bigrams in the file list. A bigram is a pair of adjacent bytes. Bytes in the database that have the high bit set are indexes (with the high bit cleared) into the bigram table. The bigram and offset-differential count coding makes these databases 20-25% smaller than the new format, but makes them not 8-bit clean. Any byte in a file name that is in the ranges used for the special codes is replaced in the database by a question mark, which not coincidentally is the shell wildcard to match a single character.

The old format therefore can not faithfully store entries with non-ASCII characters. It therefore should not be used in internationalized environments.

The output of '`locate --statistics`' will give an incorrect count of the number of filenames containing newlines or high-bit characters for old-format databases.

5.3 Newline Handling

Within the database, filenames are terminated with a null character. This is the case for both the old and the new format.

When the new database format is being used, the compression technique used to generate the database though relies on the ability to sort the list of files before they are presented to `frcode`.

If the system's sort command allows its input list of files to be separated with null characters via the '`-z`' option, this option is used and therefore `updatedb` and `locate` will both correctly handle filenames containing newlines. If the `sort` command lacks support for this, the list of files is delimited with the newline character, meaning that parts of filenames containing newlines will be incorrectly sorted. This can result in both incorrect matches and incorrect failures to match.

On the other hand, if you are using the old database format, filenames with embedded newlines are not correctly handled. There is no technical limitation which enforces this, it's just that the `bigram` program has no been updated to support lists of filenames separated by nulls.

So, if you are using the new database format (this is the default) and your system uses GNU `find`, newlines will be correctly handled at all times. Otherwise, newlines may not be correctly handled.

6 File Permissions

Each file has a set of *permissions* that control the kinds of access that users have to that file. The permissions for a file are also called its *access mode*. They can be represented either in symbolic form or as an octal number.

6.1 Structure of File Permissions

There are three kinds of permissions that a user can have for a file:

1. permission to read the file. For directories, this means permission to list the contents of the directory.
2. permission to write to (change) the file. For directories, this means permission to create and remove files in the directory.
3. permission to execute the file (run it as a program). For directories, this means permission to access files in the directory.

There are three categories of users who may have different permissions to perform any of the above operations on a file:

1. the file's owner;
2. other users who are in the file's group;
3. everyone else.

Files are given an owner and group when they are created. Usually the owner is the current user and the group is the group of the directory the file is in, but this varies with the operating system, the filesystem the file is created on, and the way the file is created. You can change the owner and group of a file by using the `chown` and `chgrp` commands.

In addition to the three sets of three permissions listed above, a file's permissions have three special components, which affect only executable files (programs) and, on some systems, directories:

1. set the process's effective user ID to that of the file upon execution (called the *setuid bit*). No effect on directories.
2. set the process's effective group ID to that of the file upon execution (called the *setgid bit*). For directories on some systems, put files created in the directory into the same group as the directory, no matter what group the user who creates them is in.
3. save the program's text image on the swap device so it will load more quickly when run (called the *sticky bit*). For directories on some systems, prevent users from removing files that they do not own in the directory; this is called making the directory *append-only*.

6.2 Symbolic Modes

Symbolic modes represent changes to files' permissions as operations on single-character symbols. They allow you to modify either all or selected parts of files' permissions, optionally based on their previous values, and perhaps on the current `umask` as well (see Section 6.2.6 [Umask and Protection], page 34).

The format of symbolic modes is:

```
[ugoa...][[+-=][rwxXstugo...]...][,...]
```

The following sections describe the operators and other details of symbolic modes.

6.2.1 Setting Permissions

The basic symbolic operations on a file's permissions are adding, removing, and setting the permission that certain users have to read, write, and execute the file. These operations have the following format:

```
users operation permissions
```

The spaces between the three parts above are shown for readability only; symbolic modes can not contain spaces.

The *users* part tells which users' access to the file is changed. It consists of one or more of the following letters (or it can be empty; see Section 6.2.6 [Umask and Protection], page 34, for a description of what happens then). When more than one of these letters is given, the order that they are in does not matter.

`u`	the user who owns the file;
`g`	other users who are in the file's group;
`o`	all other users;
`a`	all users; the same as '`ugo`'.

The *operation* part tells how to change the affected users' access to the file, and is one of the following symbols:

`+`	to add the *permissions* to whatever permissions the *users* already have for the file;
`-`	to remove the *permissions* from whatever permissions the *users* already have for the file;
`=`	to make the *permissions* the only permissions that the *users* have for the file.

The *permissions* part tells what kind of access to the file should be changed; it is zero or more of the following letters. As with the *users* part, the order does not matter when more than one letter is given. Omitting the *permissions* part is useful only with the '`=`' operation, where it gives the specified *users* no access at all to the file.

`r`	the permission the *users* have to read the file;
`w`	the permission the *users* have to write to the file;
`x`	the permission the *users* have to execute the file.

For example, to give everyone permission to read and write a file, but not to execute it, use:

```
a=rw
```

To remove write permission for from all users other than the file's owner, use:

```
go-w
```

The above command does not affect the access that the owner of the file has to it, nor does it affect whether other users can read or execute the file.

To give everyone except a file's owner no permission to do anything with that file, use the mode below. Other users could still remove the file, if they have write permission on the directory it is in.

```
go=
```

Another way to specify the same thing is:

```
og-rxw
```

6.2.2 Copying Existing Permissions

You can base part of a file's permissions on part of its existing permissions. To do this, instead of using 'r', 'w', or 'x' after the operator, you use the letter 'u', 'g', or 'o'. For example, the mode

```
o+g
```

adds the permissions for users who are in a file's group to the permissions that other users have for the file. Thus, if the file started out as mode 664 ('rw-rw-r--'), the above mode would change it to mode 666 ('rw-rw-rw-'). If the file had started out as mode 741 ('rwxr----x'), the above mode would change it to mode 745 ('rwxr--r-x'). The '-' and '=' operations work analogously.

6.2.3 Changing Special Permissions

In addition to changing a file's read, write, and execute permissions, you can change its special permissions. See Section 6.1 [Mode Structure], page 31, for a summary of these permissions.

To change a file's permission to set the user ID on execution, use 'u' in the *users* part of the symbolic mode and 's' in the *permissions* part.

To change a file's permission to set the group ID on execution, use 'g' in the *users* part of the symbolic mode and 's' in the *permissions* part.

To change a file's permission to stay permanently on the swap device, use 'o' in the *users* part of the symbolic mode and 't' in the *permissions* part.

For example, to add set user ID permission to a program, you can use the mode:

```
u+s
```

To remove both set user ID and set group ID permission from it, you can use the mode:

```
ug-s
```

To cause a program to be saved on the swap device, you can use the mode:

```
o+t
```

Remember that the special permissions only affect files that are executable, plus, on some systems, directories (on which they have different meanings; see Section 6.1 [Mode Structure], page 31). Using 'a' in the *users* part of a symbolic mode does not cause the special permissions to be affected; thus,

```
a+s
```

has *no effect*. You must use 'u', 'g', and 'o' explicitly to affect the special permissions. Also, the combinations 'u+t', 'g+t', and 'o+s' have no effect.

The '=' operator is not very useful with special permissions; for example, the mode:

```
o=t
```

does cause the file to be saved on the swap device, but it also removes all read, write, and execute permissions that users not in the file's group might have had for it.

6.2.4 Conditional Executability

There is one more special type of symbolic permission: if you use 'X' instead of 'x', execute permission is affected only if the file already had execute permission or is a directory. It affects directories' execute permission even if they did not initially have any execute permissions set.

For example, this mode:

```
a+X
```

gives all users permission to execute files (or search directories) if anyone could before.

6.2.5 Making Multiple Changes

The format of symbolic modes is actually more complex than described above (see Section 6.2.1 [Setting Permissions], page 32). It provides two ways to make multiple changes to files' permissions.

The first way is to specify multiple *operation* and *permissions* parts after a *users* part in the symbolic mode.

For example, the mode:

```
og+rX-w
```

gives users other than the owner of the file read permission and, if it is a directory or if someone already had execute permission to it, gives them execute permission; and it also denies them write permission to it file. It does not affect the permission that the owner of the file has for it. The above mode is equivalent to the two modes:

```
og+rX
og-w
```

The second way to make multiple changes is to specify more than one simple symbolic mode, separated by commas. For example, the mode:

```
a+r,go-w
```

gives everyone permission to read the file and removes write permission on it for all users except its owner. Another example:

```
u=rwx,g=rx,o=
```

sets all of the non-special permissions for the file explicitly. (It gives users who are not in the file's group no permission at all for it.)

The two methods can be combined. The mode:

```
a+r,g+x-w
```

gives all users permission to read the file, and gives users who are in the file's group permission to execute it, as well, but not permission to write to it. The above mode could be written in several different ways; another is:

```
u+r,g+rx,o+r,g-w
```

6.2.6 The Umask and Protection

If the *users* part of a symbolic mode is omitted, it defaults to '`a`' (affect all users), except that any permissions that are *set* in the system variable `umask` are *not affected*. The value of `umask` can be set using the `umask` command. Its default value varies from system to system.

Omitting the *users* part of a symbolic mode is generally not useful with operations other than '`+`'. It is useful with '`+`' because it allows you to use `umask` as an easily customizable protection against giving away more permission to files than you intended to.

As an example, if `umask` has the value 2, which removes write permission for users who are not in the file's group, then the mode:

```
+w
```

adds permission to write to the file to its owner and to other users who are in the file's group, but *not* to other users. In contrast, the mode:

```
a+w
```

ignores `umask`, and *does* give write permission for the file to all users.

6.3 Numeric Modes

File permissions are stored internally as 16 bit integers. As an alternative to giving a symbolic mode, you can give an octal (base 8) number that corresponds to the internal representation of the new mode. This number is always interpreted in octal; you do not have to add a leading 0, as you do in C. Mode 0055 is the same as mode 55.

A numeric mode is usually shorter than the corresponding symbolic mode, but it is limited in that it can not take into account a file's previous permissions; it can only set them absolutely.

The permissions granted to the user, to other users in the file's group, and to other users not in the file's group are each stored as three bits, which are represented as one octal digit. The three special permissions are also each stored as one bit, and they are as a group represented as another octal digit. Here is how the bits are arranged in the 16 bit integer, starting with the lowest valued bit:

Value in Mode	Corresponding Permission
	Other users not in the file's group:
1	Execute
2	Write
4	Read
	Other users in the file's group:
10	Execute
20	Write
40	Read
	The file's owner:
100	Execute
200	Write
400	Read
	Special permissions:
1000	Save text image on swap device
2000	Set group ID on execution
4000	Set user ID on execution

For example, numeric mode 4755 corresponds to symbolic mode '`u=rwxs,go=rx`', and numeric mode 664 corresponds to symbolic mode '`ug=rw,o=r`'. Numeric mode 0 corresponds to symbolic mode '`ugo=`'.

7 Reference

Below are summaries of the command line syntax for the programs discussed in this manual.

7.1 Invoking `find`

`find [-H] [-L] [-P] [file...] [expression]`

`find` searches the directory tree rooted at each file name *file* by evaluating the *expression* on each file it finds in the tree.

The options '`-H`', '`-L`' or '`-P`' may be specified at the start of the command line (if none of these is specified, '`-P`' is assumed). The arguments after these are a list of files or directories that should be searched.

This list of files to search is followed by a list of expressions describing the files we wish to search for. The first part of the expression is recognised by the fact that it begins with '`-`', '`(`', '`)`', '`,`', or '`!`'. Any arguments after it are the rest of the expression. If no paths are given, the current directory is used. If no expression is given, the expression '`-print`' is used.

`find` exits with status 0 if all files are processed successfully, greater than 0 if errors occur.

Three options can precede the list of path names. They determine the way that symbolic links are handled.

`-P` Never follow symbolic links (this is the default), except in the case of the '`-xtype`' predicate.

`-L` Always follow symbolic links, except in the case of the '`-xtype`' predicate.

`-H` Follow symbolic links specified in the list of paths to search, or which are otherwise specified on the command line.

If `find` would follow a symbolic link, but cannot for any reason (for example, because it has insufficient permissions or the link is broken), it falls back on using the properties of the symbolic link itself. Section 2.2.1 [Symbolic Links], page 6 for a more complete description of how symbolic links are handled.

See [Primary Index], page 49, for a summary of all of the tests, actions, and options that the expression can contain. If the expression is missing, '`-print`' is assumed.

`find` also recognizes two options for administrative use:

`--help` Print a summary of the command-line argument format and exit.

`--version`
Print the version number of `find` and exit.

7.1.1 Warning Messages

If there is an error on the `find` command line, an error message is normally issued. However, there are some usages that are inadvisable but which `find` should still accept. Under these circumstances, `find` may issue a warning message. By default, warnings are enabled only if `find` is being run interactively (specifically, if the standard input is a terminal). Warning messages can be controlled explicitly by the use of options on the command line:

`-warn` Issue warning messages where appropriate.

`-nowarn` Do not issue warning messages.

These options take effect at the point on the command line where they are specified. Therefore if you specify '`-nowarn`' at the end of the command line, you will not see warning messages for any problems occurring before that. The warning messages affected by the above options are triggered by:

– Use of the '`-d`' option which is deprecated; please use '`-depth`' instead, since the latter is POSIX-compliant.
– Use of the '`-ipath`' option which is deprecated; please use '`-iwholename`' instead.
– Specifying an option (for example '`-mindepth`') after a non-option (for example '`-type`' or '`-print`') on the command line.

The default behaviour above is designed to work in that way so that existing shell scripts which use such constructs don't generate spurious errors, but people will be made aware of the problem.

Some warning messages are issued for less common or more serious problems, and so cannot be turned off:

– Use of an unrecognised backslash escape sequence with '`-fprintf`'
– Use of an unrecognised formatting directive with '`-fprintf`'

7.2 Invoking `locate`

`locate` [*option*...] *pattern*...

`--basename`
`-b`
The specified pattern is matched against just the last component of the name of the file in the locate database. This last component is also called the "base name". For example, the base name of '`/tmp/mystuff/foo.old.c`' is '`foo.old.c`'. If the pattern contains metacharacters, it must match the base name exactly. If not, it must match part of the base name.

`--count`
`-c`
Instead of printing the matched filenames, just print the total number of matches we found.

`--database=`*path*
`-d` *path*
Instead of searching the default file name database, search the file name databases in *path*, which is a colon-separated list of database file names. You can also use the environment variable `LOCATE_PATH` to set the list of database files to search. The option overrides the environment variable if both are used. Empty elements in *path* (that is, a leading or trailing colon, or two colons in a row) are taken to stand for the default database.

`--existing`
`-e`
Only print out such names which currently exist (instead of such names which existed when the database was created). Note that this may slow down the program a lot, if there are many matches in the database. The way in which broken symbolic links are treated is affected by the '`-L`', '`-P`' and '`-H`' options.

`--follow`
`-L`
If testing for the existence of files (with the '`-e`' option), omit broken symbolic links. This is the default.

`--nofollow`
`-P`
`-H`
If testing for the existence of files (with the '`-e`' option), treat broken symbolic links count as if they were exiting files. The '`-H`' form of this option is provided purely for similarity with `find`; the use of '`-P`' is recommended over '`-H`'.

`--ignore-case`
`-i`
Ignore case distinctions in both the pattern and the file names.

`--limit=N`
`-l N` Limit the number of results printed to N. If you use the '`--count`' option, the value printed will never be larger than this limit.

`--mmap`
`-m` Accepted but does nothing. The option is supported only to provide compatibility with BSD's `locate`.

`--null`
`-0` Results are separated with the ASCII NUL character rather than the newline character. To get the full benefit of the use of this option, use the new locate database format (that is the default anyway).

`--wholename`
`-w` The specified pattern is matched against the whole name of the file in the locate database. If the pattern contains metacharacters, it must match exactly. If not, it must match part of the whole file name. This is the default behaviour.

`--regex`
`-r` Instead of using substring or shell glob matching, the pattern specified on the command line is understood to be a POSIX extended regular expression. Filenames from the locate database which match the specified regular expression are printed (or counted). If the '`-i`' flag is also given, matching is case-insensitive. Matches are performed against the whole path name, and so by default a pathname will be matched if any part of it matches the specified regular expression. The regular expression may use '`^`' or '`$`' to anchor a match at the beginning or end of a pathname.

`--stdio`
`-s` Accepted but does nothing. The option is supported only to provide compatibility with BSD's `locate`.

`--statistics`
`-S` Print some summary information for each locate database. No search is performed.

`--help` Print a summary of the options to `locate` and exit.

`--version`
Print the version number of `locate` and exit.

7.3 Invoking `updatedb`

`updatedb` [*option*...]

`--findoptions='`*OPTION*...`'`
Global options to pass on to `find`. The environment variable `FINDOPTIONS` also sets this value. Default is none.

`--localpaths='`*path*...`'`
Non-network directories to put in the database. Default is '`/`'.

`--netpaths='`*path*...`'`
Network (NFS, AFS, RFS, etc.) directories to put in the database. The environment variable `NETPATHS` also sets this value. Default is none.

`--prunepaths='`*path*...`'`
Directories to omit from the database, which would otherwise be included. The environment variable `PRUNEPATHS` also sets this value. Default is '`/tmp /usr/tmp /var/tmp /afs`'.

`--prunefs='path...'`
: File systems to omit from the database, which would otherwise be included. Note that files are pruned when a file system is reached; Any file system mounted under an undesired file system will be ignored. The environment variable `PRUNEFS` also sets this value. Default is '`nfs NFS proc`'.

`--output=dbfile`
: The database file to build. Default is system-dependent, but typically '`/usr/local/var/locatedb`'.

`--localuser=user`
: The user to search the non-network directories as, using `su`. Default is to search the non-network directories as the current user. You can also use the environment variable `LOCALUSER` to set this user.

`--netuser=user`
: The user to search network directories as, using `su`. Default is `daemon`. You can also use the environment variable `NETUSER` to set this user.

`--old-format`
: Generate a locate database in the old format, for compatibility with versions of `locate` other than GNU `locate`. Using this option means that `locate` will not be able to properly handle non-ASCII characters in filenames (that is, filenames containing characters which have the eighth bit set, such as many of the characters from the ISO-8859-1 character set).

`--help` Print a summary of the command-line argument format and exit.

`--version`
: Print the version number of `updatedb` and exit.

7.4 Invoking `xargs`

`xargs [option...] [command [initial-arguments]]`

`xargs` exits with the following status:

0	if it succeeds
123	if any invocation of the command exited with status 1-125
124	if the command exited with status 255
125	if the command is killed by a signal
126	if the command cannot be run
127	if the command is not found
1	if some other error occurred.

`--arg-file=`*inputfile*
`-a =`*inputfile*
: Read names from the file *inputfile* instead of standard input.

`--null`
`-0`
: Input filenames are terminated by a null character instead of by whitespace, and the quotes and backslash are not special (every character is taken literally). Disables the end of file string, which is treated like any other argument.

`--eof[=eof-str]`
`-e[eof-str]`
: Set the end of file string to *eof-str*. If the end of file string occurs as a line of input, the rest of the input is ignored. If *eof-str* is omitted, there is no end of file string. If this option is not given, the end of file string defaults to '`_`'.

`--help` Print a summary of the options to `xargs` and exit.

`--replace[=replace-str]`
`-i[replace-str]`
: Replace occurrences of *replace-str* in the initial arguments with names read from standard input. Also, unquoted blanks do not terminate arguments; instead, the input is split at newlines only. If *replace-str* is omitted, it defaults to '`{}`' (like for '`find -exec`'). Implies '`-x`' and '`-l 1`'.

`--max-lines[=max-lines]`
`-l[max-lines]`
: Use at most *max-lines* nonblank input lines per command line; *max-lines* defaults to 1 if omitted. Trailing blanks cause an input line to be logically continued on the next input line, for the purpose of counting the lines. Implies '`-x`'.

`--max-args=max-args`
`-n max-args`
: Use at most *max-args* arguments per command line. Fewer than *max-args* arguments will be used if the size (see the '`-s`' option) is exceeded, unless the '`-x`' option is given, in which case `xargs` will exit.

`--interactive`
`-p` Prompt the user about whether to run each command line and read a line from the terminal. Only run the command line if the response starts with '`y`' or '`Y`'. Implies '`-t`'.

`--no-run-if-empty`
`-r` If the standard input does not contain any nonblanks, do not run the command. By default, the command is run once even if there is no input.

`--max-chars=max-chars`
`-s max-chars`
: Use at most *max-chars* characters per command line, including the command and initial arguments and the terminating nulls at the ends of the argument strings.

`--verbose`
`-t` Print the command line on the standard error output before executing it.

`--version`
: Print the version number of `xargs` and exit.

`--exit`
`-x` Exit if the size (see the '`-s`' option) is exceeded.

`--max-procs=max-procs`
`-P max-procs`
: Run up to *max-procs* processes at a time; the default is 1. If *max-procs* is 0, `xargs` will run as many processes as possible at a time.

8 Security Considerations

Security considerations are important if you are using `find` or `xargs` to search for or process files that don't belong to you or over which other people have control. Security considerations relating to `locate` may also apply if you have files which you may not want others to see.

In general, the most severe forms of security problems affecting `find` and related programs are where third parties can bring about a situation where those programs allow them to do something they would normally not be able to do. This is called *privilege elevation*. This might include deleting files they would not normally be able to delete. It is also common for the system to periodically invoke `find` for housekeeping purposes. These invocations of `find` are particularly problematic from a security point of view as these are often invoked by the superuser and search the whole file system hierarchy. The severity of any associated problem depends on what the system is going to do with the output of `find`.

8.1 Levels of Risk

There are some security risks inherent in the use of `find`, `xargs` and (to a lesser extent) `locate`. The severity of these risks depends on what sort of system you are using:

High risk Multi-user systems where you do not control (or trust) the other users, and on which you execute `find`, including areas where those other users can manipulate the filesystem (for example beneath '`/home`' or '`/tmp`').

Medium Risk
Systems where the actions of other users can create filenames chosen by them, but to which they don't have access while `find` is being run. This access might include leaving programs running (shell background jobs, `at` or `cron` tasks, for example). On these sorts of systems, carefully written commands (avoiding use of '`-print`' for example) should not expose you to a high degree of risk. Most systems fall into this category.

Low Risk Systems to which untrusted parties do not have access, cannot create filenames of their own choice (even remotely) and which contain no security flaws which might enable an untrusted third party to gain access. Most systems do not fall into this category because there are many ways in which external parties can affect the names of files that are created on your system. The system on which I am writing this for example automatically downloads software updates from the Internet; the names of the files in which these updates exist are chosen by third parties[1].

In the discussion above, "risk" denotes the likelihood that someone can cause `find`, `xargs`, `locate` or some other program which is controlled by them to do something you did not intend. The levels of risk suggested do not take any account of the consequences of this sort of event. That is, if you operate a "low risk" type system, but the consequences of a security problem are disastrous, then you should still give serious thought to all the possible security problems, many of which of course will not be discussed here – this section of the manual is intended to be informative but not comprehensive or exhaustive.

If you are responsible for the operation of a system where the consequences of a security problem could be very important, you should do two things:-

1. Define a security policy which defines who is allowed to do what on your system
2. Seek competent advice on how to enforce your policy, detect breaches of that policy, and take account of any potential problems that might fall outside the scope of your policy

[1] Of course, I trust these parties to a large extent anyway, because I install software provided by them; I choose to trust them in this way, and that's a deliberate choice

8.2 Security Considerations for find

Some of the actions `find` might take have a direct effect; these include `-exec` and `-delete`. However, it is also common to use `-print` explicitly or implicitly, and so if `find` produces the wrong list of filenames, that can also be a security problem; consider the case for example where `find` is producing a list of files to be deleted.

We normally assume that the `find` command line expresses the file selection criteria and actions that the user had in mind – that is, the command line is "trusted" data.

From a security analysis point of view, the output of `find` should be correct; that is, the output should contain only the names of those files which meet the user's criteria specified on the command line. This applies for the `-exec` and `-delete` actions; one can consider these to be part of the output.

On the other hand, the contents of the filesystem can be manipulated by other people, and hence we regard this as "untrusted" data. This implies that the `find` command line is a filter which converts the untrusted contents of the filesystem into a correct list of output files.

The filesystem will in general change while `find` is searching it; in fact, most of the potential security problems with `find` relate to this issue in some way.

Race conditions are a general class of security problem where the relative ordering of actions taken by `find` (for example) and something else are important[2] .

Typically, an attacker might move or rename files or directories in the hope that an action might be taken against a a file which was not normally intended to be affected. Alternatively, this sort of attack might be intended to persuade `find` to search part of the filesystem which would not normally be included in the search (defeating the `-prune` action for example).

8.2.1 Changing the Current Working Directory

As find searches the file system, it finds subdirectories and then searches within them by changing its working directory. First, `find` notices a subdirectory. It then decides if that subdirectory meets the criteria for being searched; that is, any '`-xdev`' or '`-prune`' expressions are taken into account. The `find` program will then change working directory and proceed to search the directory.

A race condition attack might take the form that once the checks relevant to '`-xdev`' and '`-prune`' have been done, an attacker might rename the directory that was being considered, and put in its place a symbolic link that actually points somewhere else.

The idea behind this attack is to fool `find` into going into the wrong directory. This would leave `find` with a working directory chosen by an attacker, bypassing any protection apparently provided by '`-xdev`' and '`-prune`', and any protection provided by being able to *not* list particular directories on the `find` command line. This form of attack is particularly problematic if the attacker can predict when the `find` command will be run, as is the case with `cron` tasks for example.

GNU `find` has specific safeguards to prevent this general class of problem. The exact form of these safeguards depends on the properties of your system.

8.2.1.1 O_NOFOLLOW

If your system supports the O_NOFOLLOW flag[3] to the `open(2)` system call, `find` uses it when safely changing directory. The target directory is first opened and then `find` changes working directory with the `fchdir()` system call. This ensures that symbolic links are not followed, preventing the sort of race condition attack in which use is made of symbolic links.

[2] This is more or less the definition of the term "race condition"

[3] GNU/Linux (kernel version 2.1.126 and later) and FreeBSD (3.0-CURRENT and later) support this

If for any reason this approach does not work, `find` will fall back on the method which is normally used if O_NOFOLLOW is not supported.

You can tell if your system supports O_NOFOLLOW by running

```
find --version
```

This will tell you the version number and which features are enabled. For example, if I run this on my system now, this gives:

```
GNU find version 4.2.18-CVS
Features enabled: D_TYPE O_NOFOLLOW(enabled)
```

Here, you can see that I am running a version of find which was built from the development (CVS) code prior to the release of findutils-4.2.18, and that the D_TYPE and O_NOFOLLOW features are present. O_NOFOLLOW is qualified with “enabled”. This simply means that the current system seems to support O_NOFOLLOW. This check is needed because it is possible to build find on a system that defines O_NOFOLLOW and then run it on a system that ignores the O_NOFOLLOW flag. We try to detect such cases at startup by checking the operating system and version number; when this happens you will see “O_NOFOLLOW(disabled)” instead.

8.2.1.2 Systems without O_NOFOLLOW

The strategy for preventing this type of problem on systems that lack support for the O_NOFOLLOW flag is more complex. Each time `find` changes directory, it examines the directory it is about to move to, issues the `chdir()` system call, and then checks that it has ended up in the subdirectory it expected. If not, an error message is issued and `find` exits immediately. This method prevents filesystem manipulation attacks from persuading `find` to search parts of the filesystem it did not intend. However, we heve to take special steps in order not to unneccessarily conclude that there is a problem with any “automount” mount points.

8.2.1.3 Working with automounters

Where an automounter is in use it can be the case that the use of the `chdir()` system call can itself cause a new filesystem to be mounted at that point. On systems that do not support O_NOFOLLOW, this will cause `find`’s security check to fail.

However, this does not normally represent a security problem (since the automounter configuration is normally set up by the system administrator). Therefore, if the `chdir()` sanity check fails, `find` will check to see if a new filesystem has been mounted at the current directory; if so, `find` will issue a warning message and continue.

To make this solution work, `find` reads the list of mounted filesystems at startup, and again when the sanity check fails. It compares the two lists to find out if the directory it has moved into has just been mounted.

8.2.1.4 Problems with dead NFS servers

Examining every mount point on the system has a downside too. In general, `find` will be used to search just part of the filesystem. However, `find` examines every mount point. If the system has a filesystem mounted on an unresponsive NFS server, `find` will hang, waiting for the NFS server to respond. Worse, it does this even if the affected mount point is not within the directory tree that find would have searched anyway.

This is very unfortunate. However, this problem only affects systems that have no support for O_NOFOLLOW. As far as I can tell, it is not possible on such systems to fix all three problems (the race condition, the false-alarm at automount mount points, and the hang at startup if there is a dead NFS server) at once. If you have some ideas about how `find` could do this better, please send email to the `bug-findutils@gnu.org` mailing list.

8.2.2 Race Conditions with -exec

The '`-exec`' action causes another program to be run. It is passed the name of the file which is being considered at the time. The invoked program will then - normally - perform some action on that file. Once again, there is a race condition which can be exploited here. We shall take as a specific example the command

```
find /tmp -path /tmp/umsp/passwd -exec /bin/rm
```

In this simple example, we are identifying just one file to be deleted and invoking `/bin/rm` to delete it. A problem exists because there is a time gap between the point where `find` decides that it needs to process the '`-exec`' action and the point where the `/bin/rm` command actually issues the `unlink()` system call. Within this time period, an attacker can rename the '`/tmp/umsp`' directory, replacing it with a symbolic link to '`/etc`'. There is no way for `/bin/rm` to determine that it is working on the same file that `find` had in mind. Once the symbolic link is in place, the attacker has persuaded `find` to cause the deletion of the '`/etc/passwd`' file, which is not the effect intended by the command which was actually invoked.

One possible defence against this type of attack is to modify the behaviour of '`-exec`' so that the `/bin/rm` command is run with the argument '`./passwd`' and a suitable choice of working directory. This would allow the normal sanity check that `find` performs to protect against this form of attack too. Unfortunately, this strategy cannot be used as the POSIX standard specifies that the current working directory for commands invoked via '`-exec`' must be the same as the current working directory from which `find` was invoked. This means that the '`-exec`' action is inherently insecure and can't be fixed.

GNU `find` implements a more secure variant of the '`-exec`' action, '`-execdir`'. The '`-execdir`' action ensures that it is not necessary to dereference subdirectories to process target files. The current directory used to invoke programs is the same as the directory in which the file to be processed exists ('`/tmp/umsp`' in our example, and only the basename of the file to be processed is passed to the invoked command, with a '`./`' prepended (giving '`./passwd`' in our example).

The '`-execdir`' action refuses to do anything if the current directory is included in the *$PATH* environment variable. This is necessary because '`-execdir`' runs programs in the same directory in which it finds files – in general, such a directory might be writable by untrusted users. For similar reasons, '`-execdir`' does not allow '`{}`' to appear in the name of the command to be run.

8.2.3 Race Conditions with -print and -print0

The '`-print`' and '`-print0`' actions can be used to produce a list of files matching some criteria, which can then be used with some other command, perhaps with `xargs`. Unfortunately, this means that there is an unavoidable time gap between `find` deciding that one or more files meet its criteria and the relevant command being executed. For this reason, the '`-print`' and '`-print0`' actions are just as insecure as '`-exec`'.

In fact, since the construction

```
find ....   -print | xargs ....
```

does not cope correctly with newlines or other "white space" in filenames, and copes poorly with filenames containing quotes, the '`-print`' action is less secure even than '`-print0`'.

8.3 Security Considerations for `xargs`

The description of the race conditions affecting the '`-print`' action of `find` shows that `xargs` cannot be secure if it is possible for an attacker to modify a filesystem after `find` has started but before `xargs` has completed all its actions.

However, there are other security issues that exist even if it is not possible for an attacker to have access to the filesystem in real time. Firstly, if it is possible for an attacker to create files with names of their own choice on the filesystem, then `xargs` is insecure unless the '`-0`' option is used. If a file with the name '`/home/someuser/foo/bar\n/etc/passwd`' exists (assume that '`\n`' stands for a newline character), then `find ... -print` can be persuaded to print three separate lines:

```
/home/someuser/foo/bar

/etc/passwd
```

If it finds a blank line in the input, `xargs` will ignore it. Therefore, if some action is to be taken on the basis of this list of files, the '`/etc/passwd`' file would be included even if this was not the intent of the person running find. There are circumstances in which an attacker can use this to their advantage. The same consideration applies to filenames containing ordinary spaces rather than newlines, except that of course the list of filenames will no longer contain an "extra" newline.

This problem is an unavoidable consequence of the default behaviour of the `xargs` command, which is specified by the POSIX standard. The only ways to avoid this problem are either to avoid all use of `xargs` in favour for example of '`find -exec`' or (where available) '`find -execdir`', or to use the '`-0`' option, which ensures that `xargs` considers filenames to be separated by ASCII NUL characters rather than whitespace. However, useful though this option is, the POSIX standard does not make it mandatory.

8.4 Security Considerations for `locate`

It is fairly unusual for the output of `locate` to be fed into another command. However, if this were to be done, this would raise the same set of security issues as the use of '`find ... -print`'. Although the problems relating to whitespace in filenames can be resolved by using `locate`'s '`-0`' option, this still leaves the race condition problems associated with '`find ... -print0`'. There is no way to avoid these problems in the case of `locate`.

8.5 Summary

Where untrusted parties can create files on the system, or affect the names of files that are created, all uses for `find`, `locate` and `xargs` have known security problems except the following:

Informational use only
: Uses where the programs are used to prepare lists of filenames upon which no further action will ever be taken.

-delete
: Use of the '`-delete`' action to delete files which meet specified criteria

-execdir
: Use of the '`-execdir`' action where the `PATH` environment variable contains directories which contain only trusted programs.

9 Error Messages

This section describes some of the error messages you might get from `find`, `xargs`, or `locate`, explains them and in some cases provides advice as to what you should do about this.

This manual is written in English. The GNU findutils software features translated error messages for many languages. For this reason where possible we try to make the error messages produced by the programs self-explanatory. This approach avoids asking people to figure out which English-language error message the test they actually saw might correspond to. Error messages which are self-explanatory will not normally be described or discussed in this document. For those messages which are discussed in this document, only the English-language version of the message will be listed.

9.1 Error Messages From find

'`invalid predicate '-foo'`'

This means that the `find` command line included something that started with a dash or other special character. The `find` program tried to interpret this as a test, action or option, but didn't recognise it. If you intended it to be a test, check what you specified against the documentation. If, on the other hand, the string is the name of a file which has been expanded from a wildcard (for example because you have a '`*`' on the command line), consider using '`./*`' or just '`.`' instead.

'`unexpected extra predicate`'

This usually happens if you have an extra bracket on the command line (for example '`find . -print \)`').

'`Warning: filesystem /path/foo has recently been mounted`'
'`Warning: filesystem /path/foo has recently been unmounted`'

These messages might appear when `find` moves into a directory and finds that the device number and inode are different to what it expected them to be. If the directory `find` has moved into is on an NFS filesystem, it will not issue this message, because `automount` frequently mounts new filesystems on directories as you move into them (that is how it knows you want to use the filesystem). So, if you do see this message, be wary – `automount` may not have been responsible. Consider the possibility that someone else is manipulating the filesystem while `find` is running. Some people might do this in order to mislead `find` or persuade it to look at one set of files when it thought it was looking at another set.

'`/path/foo changed during execution of find (old device number 12345, new device number 6789, filesystem type is <whatever>) [ref XXX]`'

This message is issued when `find` changes directory and ends up somewhere it didn't expect to be. This happens in one of two circumstances. Firstly this happens when "automount" does its thing on a system where `find` doesn't know how to determine what the current set of mounted filesystems is

Secondly, this can happen when the device number of a directory appears to change during a change of current directory, but `find` is moving up the filesystem hierarchy rather than down it. In order to prevent `find` wandering off into some unexpected part of the filesystem, we stop it at this point.

'`Don't know how to use getmntent() to read '/etc/mtab'. This is a bug.`'

This message is issued when a problem similar to the above occurs on a system where `find` doesn't know how to figure out the current list of mount points. Ask for help on `bug-findutils@gnu.org`.

'`/path/foo/bar changed during execution of find (old inode number 12345, new inode number 67893, filesystem type is <whatever>) [ref XXX]"),`'

: This message is issued when `find` changes directory and discovers that the inode number of that directory once it's got there is different to the inode number that it obtained when it examined the directory some time previously. This normally means that while `find` has been deep in a directory hierarchy doing something time consuming, somebody has moved the one of the parent directories to another location in the same filesystem. This may have been done maliciously, or may not. In any case, `find` stops at this point in order to avoid traversing parts of the filesystem that it wasn't intended to. You can use `ls -li` or `find /path -inum 12345 -o -inum 67893` to find out more about what has happened.

'`sanity check of the fnmatch() library function failed.`'

: Please submit a bug report. You may well be asked questions about your system, and if you compiled the `findutils` code yourself, you should keep your copy of the build tree around. The likely explanation is that your system has a buggy implementation of `fnmatch` that looks enough like the GNU version to fool `configure`, but which doesn't work properly.

'`cannot fork`'

: This normally happens if you use the `-exec` action or a something similar (`-ok` and so forth) but the system has run out of free process slots. This is either because the system is very busy and the system has reached its maximum process limit, or because you have a resource limit in place and you've reached it. Check the system for runaway processes (if `ps` still works). Some process slots are normally reserved for use by '`root`'.

'`some-program terminated by signal 99`'

: Some program which was launched via `-exec` or similar was killed with a fatal signal. This is just an advisory message.

9.2 Error Messages From xargs

'`environment is too large for exec`'

: This message means that you have so many environment variables set (or such large values for them) that there is no room within the system-imposed limits on program command-line argument length to invoke any program. I'm sure you did this deliberately. Please try unsetting some environment variables, or exiting the current shell.

'`can not fit single argument within argument list size limit`'

: You are using the '`-i`' option and `xargs` doesn't have enough space to build a command line because it has read in a really large item and it doesn't fit. You can probably work around this problem with the '`-s`' option, but the default size is pretty large. You must be trying pretty hard to break `xargs`.

'`cannot fork`'

: See the description of the similar message for `find`.

'`<program>: exited with status 255; aborting`'

: When a command run by `xargs` exits with status 255, `xargs` is supposed to stop. If this is not what you intended, wrap the program you are trying to invoke in a shell script which doesn't return status 255.

'`<program>: terminated by signal 99`'

: See the description of the similar message for `find`.

9.3 Error Messages From locate

'`warning: database '/usr/local/var/locatedb' is more than 8 days old`'
: The `locate` program relies on a database which is periodically built by the `updatedb` program. That hasn't happened in a long time. To fix this problem, run `updatedb` manually. This can often happen on systems that are generally not left on, so the periodic "cron" task which normally does this doesn't get a chance to run.

'`locate database '/usr/local/var/locatedb' is corrupt or invalid`'
: This should not happen. Re-run `updatedb`. If that works, but `locate` still produces this error, run `locate --version` and `updatedb --version`. These should produce the same output. If not, you are using a mixed toolset; check your '`$PATH`' environment variable and your shell aliases (if you have any). If both programs claim to be GNU versions, this is a bug; all versions of these programs should interoperate without problem. Ask for help on `bug-findutils@gnu.org`.

9.4 Error Messages From updatedb

The `updatedb` program (and the programs it invokes) do issue error messages, but none of them seem to me to be candidates for guidance. If you are having a problem understanding one of these, ask for help on `bug-findutils@gnu.org`.

find Primary Index

This is a list of all of the primaries (tests, actions, and options) that make up `find` expressions for selecting files. See Section 1.3 [find Expressions], page 2, for more information on expressions.