10

Configuration: Starting on the Right Foot

> Let us watch well our beginnings, and results will manage themselves.
>
> —Alexander Clark

Under Unix, programs can communicate with their environment in a rich variety of ways. It's convenient to divide these into (a) startup-environment queries and (b) interactive channels. In this chapter, we'll focus primarily on startup-environment queries. The next chapter will discuss interactive channels.

10.1 What Should Be Configurable?

Before plunging into the details of different kinds of program configuration, we should ask a high-level question: What things should be configurable?

The gut-level Unix answer is "everything". The Rule of Separation that we discussed in Chapter 1 encourages Unix programmers to build mechanism and defer policy decisions outward toward the user wherever possible. While this tends to produce programs that are powerful and rewarding for expert users, it also tends to produce interfaces that overwhelm novices and casual users with a surfeit of choices, and with configuration files sprouting like weeds.

Unix programmers aren't going to be cured of their tendency to design for their peers and the most sophisticated users any time soon (we'll grapple a bit with the

question of whether such a change would actually be desirable in Chapter 20). So it's perhaps more useful to invert the question and ask "What things should *not* be configurable?" Unix practice does offer some guidelines on this.

First, *don't provide configuration switches for what you can reliably detect automatically*. This is a surprisingly common mistake. Instead, look for ways to eliminate configuration switches by autodetection, or by trying alternative methods at runtime until one succeeds. If this strikes you as inelegant or too expensive, ask yourself if you haven't fallen into premature optimization.

> One of the nicest examples of autodetection I experienced was when Dennis Ritchie and I were porting Unix to the Interdata 8/32. This was a big-endian machine, and we had to generate data for that machine on a PDP-11, write a magnetic tape, and then load the magnetic tape on the Interdata. A common error was to forget to twiddle the byte order; a checksum error showed you that you had to unmount, remount again on the PDP-11, regenerate the tape, unmount, and remount. Then one day Dennis hacked the Interdata tape reader program so that if it got a checksum error it rewound the tape, toggled 'byte flip' switch and reread it. A second checksum error would kill the load, but 99% of the time it just read the tape and did the right thing. Our productivity shot up, and we pretty much ignored tape byte order from that point on.
>
> —Steve Johnson

A good rule of thumb is this: Be adaptive unless doing so costs you 0.7 seconds or more of latency. 0.7 seconds is a magic number because, as Jef Raskin discovered while designing the Canon Cat, humans are almost incapable of noticing startup latency shorter than that; it gets lost in the mental overhead of changing the focus of attention.

Second, *users should not see optimization switches*. As a designer, it's *your* job to make the program run economically, not the user's. The marginal gains in performance that a user might collect from optimization switches are usually not worth the interface-complexity cost.

> File-format nonsense (record length, blocking factor, etc) was blessedly eschewed by Unix, but the same kind of thing has roared back in excess configuration goo. KISS became MICAHI: make it complicated and hide it.
>
> —Doug McIlroy

Finally, *don't do with a configuration switch what can be done with a script wrapper or a trivial pipeline*. Don't put complexity inside your program when you

can easily enlist other programs to help get the work done. (Recall our discussion in Chapter 7 of why *ls*(1) does not have a built-in pager, or an option to invoke it).

Here are some more general questions to consider whenever you find yourself thinking about adding a configuration option:

- Can I leave this feature out? Why am I fattening the manual and burdening the user?
- Could the program's normal behavior be changed in an innocuous way that would make the option unnecessary?
- Is this option merely cosmetic? Should I be thinking less about how to make the user interface configurable and more about how to make it right?
- Should the behavior enabled by this option be a separate program instead?

> Unless it is done very carefully, the addition of an on/off configuration option can lead to a need to double the amount of testing. Since in practice one never does double the amount of testing, the practical effect is to reduce the amount of testing that any given configuration receives. Ten options leads to 1024 times as much testing, and pretty soon you are talking real reliability problems.
>
> —Steve Johnson

10.2 Where Configurations Live

Classically, a Unix program can look for control information in five places in its startup-time environment:

- Run-control files under `/etc` (or at fixed location elsewhere in system-land).
- System-set environment variables.
- Run-control files (or 'dotfiles') in the user's home directory. (See Chapter 3 for a discussion of this important concept, if it is unfamiliar.)
- User-set environment variables.
- Switches and arguments passed to the program on the command line that invoked it.

These queries are usually done in the order listed above. That way, later (more local) settings override earlier (more global) ones. Settings found earlier can help the program compute locations for later retrievals of configuration data.

When thinking about which mechanism to use to pass configuration data to a program, bear in mind that good Unix practice demands using whichever one most closely matches the expected lifetime of the preference. Thus: for preferences which are very likely to change between invocations, use command-line switches. For preferences which change seldom, but that should be under individual user control, use a run-control file in the user's home directory. For preference information that needs to be set site-wide by a system administrator and *not* changed by users, use a run-control file in system space.

We'll discuss each of these places in more detail, then examine some case studies.

10.3 Run-Control Files

A run-control file is a file of declarations or commands associated with a program that it interprets on startup. If a program has site-specific configuration shared by all users at a site, it will often have a run control file under the `/etc` directory. (Some Unixes have an `/etc/conf` subdirectory that collects such data.)

User-specific configuration information is often carried in a hidden run-control file in the user's home directory. Such files are often called 'dotfiles' because they exploit the Unix convention that a filename beginning with a dot is normally invisible to directory-listing tools.[1]

Programs can also have run-control or dot directories. These group together several configuration files that are related to the program, but that are most conveniently treated separately (perhaps because they relate to different subsystems of the program, or have differing syntaxes).

Whether file or directory, convention now dictates that the location of the run-control information has the same basename as the executable that reads it. An older convention still common among system programs uses the executable's name with the suffix 'rc' for 'run control'.[2] Thus, if you write a program called 'seekstuff' that has both site-wide and user-specific configuration, an experienced Unix user would expect to find the former at `/etc/seekstuff` and the latter at `.seekstuff` in the user's home directory; but it would be unsurprising if the locations were

1. To make dotfiles visible, use the `-a` option of *ls*(1).

2. The 'rc' suffix goes back to Unix's grandparent, CTSS. It had a command-script feature called "runcom". Early Unixes used 'rc' for the name of the operating system's boot script, as a tribute to CTSS runcom.

`/etc/seekstuffrc` and `.seekstuffrc`, especially if seekstuff were a system utility of some sort.

In Chapter 5 we described a somewhat different set of design rules for textual data file formats, and discussed how to optimize for different weightings of interoperability, transparency and transaction economy. Run-control files are typically only read once at program startup and not written; economy is therefore usually not a major concern. Interoperability and transparency both push us toward textual formats designed to be read by human beings and modified with an ordinary text editor.

While the semantics of run-control files are of course completely program dependent, there are some design rules about run-control syntax that are widely observed. We'll describe those next; but first we'll describe an important exception.

If the program is an interpreter for a language, then it is expected to be simply a file of commands in the syntax of that language, to be executed at startup. This is an important rule, because Unix tradition strongly encourages the design of all kinds of programs as special-purpose languages and minilanguages. Well-known examples with dotfiles of this kind include the various Unix command shells and the *Emacs* programmable editor.

(One reason for this design rule is the belief that special cases are bad news—thus, that any switch that changes the behavior of a language should be settable from within the language. If as a language designer you find that you *cannot* express all the startup settings of a language in the the language itself, a Unix programmer would say you have a design problem—which is what you should be fixing, rather than devising a special-case run-control syntax.)

This exception aside, here are the normal style rules for run control syntaxes. Historically, they are patterned on the syntax of Unix shells:

1. *Support explanatory comments, and lead them with #.* The syntax should also ignore whitespace before #, so that comments on the same line as configuration directives are supported.

2. *Don't make insidious whitespace distinctions.* That is, treat runs of spaces and tabs, syntactically the same as a single space. If your directive format is line-oriented, it is good form to ignore trailing spaces and tabs on lines. The metarule is that the interpretation of the file should not depend on distinctions a human eye can't see.

3. *Treat multiple blank lines and comment lines as a single blank line*. If the input format uses blank lines as separators between records, you probably want to ensure that a comment line does not end a record.

4. *Lexically treat the file as a simple sequence of whitespace-separated tokens, or lines of tokens.* Complicated lexical rules are hard to learn, hard to remember, and hard for humans to parse. Avoid them.

5. *But, support a string syntax for tokens with embedded whitespace.* Use single-quote or double-quote as balanced delimiters. If you support both, beware of giving different semantics as they have in shell; this is a well-known source of confusion.

6. *Support a backslash syntax for embedding unprintable and special characters in strings*. The standard pattern for this is the backslash-escape syntax supported by C compilers. Thus, for example, it would be quite surprising if the string `"a\tb"` were not interpreted as a character 'a', followed by a tab, followed by the character 'b'.

Some aspects of shell syntax, on the other hand, should *not* be emulated in run-control syntaxes—at least not without a good and specific reason. The shell's baroque quoting and bracketing rules, and its special metacharacters for wildcards and variable substitution, both fall in this category.

It bears repeating that the point of these conventions is to reduce the amount of novelty that users have to cope with when they read and edit the run-control file for a program they have never seen before. Therefore, if you have to break the conventions, try to do so in a way that makes it visually obvious that you have done so, document your syntax with particular care, and (most importantly) design it so it's easy to pick up by example.

These standard style rules only describe conventions about tokenizing and comments. The names of run-control files, their higher-level syntax, and the semantic interpretation of the syntax are usually application-specific. There are a very few exceptions to this rule, however; one is dotfiles which have become 'well-known' in the sense that they routinely carry information used by a whole class of applications. Sharing run-control file formats in this way reduces the amount of novelty users have to cope with.

Of these, probably the best established is the `.netrc` file. Internet client programs that must track host/password pairs for a user can usually get them from the `.netrc` file, if it exists.

10.3.1 Case Study: The `.netrc` File

The `.netrc` file is a good example of the standard rules in action. An example, with the passwords changed to protect the innocent, is in Example 10.1.

Example 10.1: A `.netrc` example.

```
# FTP access to my Web host
machine unix1.netaxs.com
        login esr
        password joesatriani

# My main mailserver at Netaxs
machine imap.netaxs.com
        login esr
        password jeffbeck

# Auxiliary IMAP maildrop at CCIL
machine imap.ccil.org
    login esr
    password marcbonilla

# Auxiliary POP maildrop at CCIL
machine pop3.ccil.org
    login esr
    password ericjohnson

# Shell account at CCIL
machine locke.ccil.org
    login esr
    password stevemorse
```

Observe that this format is easy to parse by eyeball even if you've never seen it before; it's a set of machine/login/password triples, each of which describes an account on a remote host. This kind of transparency is important—much more important, actually, than the time economy of faster interpretation or the space economy of a more compact and cryptic file format. It economizes the far more valuable resource that is *human* time, by making it likely that a human being will be able to read and modify the format without having to read a manual or use a tool less familiar than a plain old text editor.

Observe also that this format is used to supply information for multiple services—an advantage, because it means sensitive password information need only be stored in one place. The `.netrc` format was designed for the original Unix FTP client program. It's used by all FTP clients, and also understood by some telnet clients, and by *smbclient*(1) command-line tool, and by the *fetchmail* program. If you are writing an Internet client that must do password authentication through remote logins, the Rule of Least Surprise demands that it use the contents of `.netrc` as defaults.

10.3.2 Portability to Other Operating Systems

System-wide run-control files are a design tactic that can be used on almost any operating system, but dotfiles are rather more difficult to map to a non-Unix environment. The critical thing missing from most non-Unix operating systems is true multi-user capability and the notion of a per-user home directory. DOS and Windows versions up to ME (including 95 and 98), for example, completely lack any such notion; all configuration information has to be stored either in system-wide run control files at a fixed location, the Windows registry, or configuration files in the same directory a program is run from. Windows NT has some notion of per-user home directories (which made its way into Windows 2000 and XP), but it is only poorly supported by the system tools.

10.4 Environment Variables

When a Unix program starts up, the environment accessible to it includes a set of name to value associations (names and values are both strings). Some of these are set manually by the user; others are set by the system at login time, or by your shell or terminal emulator (if you're running one). Under Unix, environment variables tend to carry information about file search paths, system defaults, the current user ID and process number, and other key bits of information about the runtime einvironment of programs. At a shell prompt, typing **`set`** followed by a newline will list all currently defined shell variables.

In C and C++ these values can be queried with the library function *getenv*(3). Perl and Python initialize environment-dictionary objects at startup. Other languages generally follow one of these two models.

10.4.1 System Environment Variables

There are a number of well-known environment variables you can expect to find defined on startup of a program from the Unix shell. These (especially `HOME`) will often need to be evaluated *before* you read a local dotfile.

`USER`
: Login name of the account under which this session is logged in (BSD convention).

`LOGNAME`
: Login name of the account under which this session is logged in (System V convention).

HOME
Home directory of the user running this session.

COLUMNS
The number of character-cell columns on the controlling terminal or terminal-emulator window.

LINES
The number of character-cell rows on the controlling terminal or terminal-emulator window.

SHELL
The name of the user's command shell (often used by shellout commands).

PATH
The list of directories that the shell searches when looking for executable commands to match a name.

TERM
Name of the terminal type of the session console or terminal emulator window (see the terminfo case study in Chapter 6 for background). TERM is special in that programs to create remote sessions over the network (such as *telnet* and *ssh*) are expected to pass it through and set it in the remote session.

(This list is representative, but not exhaustive.)

The `HOME` variable is especially important, because many programs use it to find the calling user's dotfiles (others call some functions in the C runtime library to get the calling user's home directory).

Note that some or all of these system environment variables may *not* be set when a program is started by some other method than a shell spawn. In particular, daemon listeners on a TCP/IP socket often don't have these variables set—and if they do, the values are unlikely to be useful.

Finally, note that there is a tradition (exemplified by the `PATH` variable) of using a colon as a separator when an environment variable must contain multiple fields, especially when the fields can be interpreted as a search path of some sort. Note that some shells (notably `bash` and `ksh`) *always* interpret colon-separated fields in an environment variable as filenames, which means in particular that they expand ~ in these fields to the user's home directory.

10.4.2 User Environment Variables

Although applications are free to interpret environment variables outside the system-defined set, it is nowadays fairly unusual to actually do so. Environment values are not really suitable for passing structured information into a program (though it can in principle be done via parsing of the values). Instead, modern Unix applications tend to use run-control files and dotfiles.

There are, however, some design patterns in which user-defined environment variables can be useful:

Application-independent preferences that need to be shared by a large number of different programs. This set of 'standard' preferences changes only slowly, because lots of different programs need to recognize each one before it becomes useful.[3] Here are the standard ones:

EDITOR
: The name of the user's preferred editor (often used by shellout commands).[4]

MAILER
: The name of the user's preferred mail user agent (often used by shellout commands).

PAGER
: The name of the user's preferred program for browsing plaintext.

BROWSER
: The name of the user's preferred program for browsing Web URLs. This one, as of 2003, is still very new and not yet widely implemented.

10.4.3 When to Use Environment Variables

What both user and system environment variables have in common is that it would be annoying to have to replicate the information they contain in a large number of application run-control files, and extremely annoying to have to change that information

3. Nobody knows a really graceful way to represent this sort of distributed preference data; environment variables probably are not it, but all the known alternatives have equally nasty problems.

4. Actually, most Unix programs first check `VISUAL`, and only if that's not set will they consult `EDITOR`. That's a relic from the days when people had different preferences for line-oriented editors and visual editors.

everywhere when your preference changes. Typically, the user sets these variables in his or her shell session startup file.

A value varies across several contexts that share dotfiles, or a parent needs to pass information to multiple child processes. Some pieces of start-up information are expected to vary across several contexts in which the calling user would share common run-control files and dotfiles. For a system-level example, consider several shell sessions open through terminal emulator windows on an X desktop. They will all see the same dotfiles, but might have different values of `COLUMNS`, `LINES`, and `TERM`. (Old-school shell programming used this method extensively; makefiles still do.)

A value varies too often for dotfiles, but doesn't change on every startup. A user-defined environment variable may (for example) be used to pass a file-system or Internet location that is the root of a tree of files that the program should play with. The CVS version-control system interprets the variable `CVSROOT` this way, for example. Several newsreader clients that fetch news from servers using the NNTP protocol interpret the variable `NNTPSERVER` as the location of the server to query.

A process-unique override needs to be expressed in a way that doesn't require the command-line invocation to be changed. A user-defined environment variable can be useful for situations in which, for whatever reason, it would be inconvenient to have to change an application dotfile or supply command-line options (perhaps it is expected that the application will normally be used inside a shell wrapper or within a makefile). A particularly important context for this sort of use is debugging. Under Linux, for example, manipulating the variable `LD_LIBRARY_PATH` associated with the *ld*(1) linking loader enables you to change where libraries are loaded from—perhaps to pick up versions that do buffer-overflow checking or profiling.

In general, a user-defined environment variable can be an effective design choice when the value changes often enough to make editing a dotfile each time inconvenient, but not necessarily every time (so always setting the location with a command-line option would also be inconvenient). Such variables should typically be evaluated *after* a local dotfile and be permitted to override settings in it.

There is one traditional Unix design pattern that we do not recommend for new programs. Sometimes, user-set environment variables are used as a lightweight substitute for expressing a program preference in a run-control file. The venerable *nethack*(1) dungeon-crawling game, for example, reads a `NETHACKOPTIONS` environment variable for user preferences. This is an old-school technique; modern practice would lean toward parsing them from a `.nethack` or `.nethackrc` run-control file.

The problem with the older style is that it makes tracking where your preference information lives more difficult than it would be if you knew the program had a run-control file under your home directory. Environment variables can be set anywhere in several different shell run-control files—under Linux these are likely to include `.profile`, `.bash_profile`, and `.bashrc` at least. These files are cluttered

and fragile things, so as the code overhead of having an option-parser has come to seem less significant preference information has tended to migrate out of environment variables into dotfiles.

10.4.4 Portability to Other Operating Systems

Environment variables have only very limited portability off Unix. Microsoft operating systems have an environment-variable feature modeled on that of Unix, and use a `PATH` variable as Unix does to set the binary search path, but most of other variables that Unix shell programmers take for granted (such as process ID or current working directory) are not supported. Other operating systems (including classic MacOS) generally do not have any local equivalent of environment variables.

10.5 Command-Line Options

Unix tradition encourages the use of command-line switches to control programs, so that options can be specified from scripts. This is especially important for programs that function as pipes or filters. Three conventions for how to distinguish command-line options from ordinary arguments exist; the original Unix style, the GNU style, and the X toolkit style.

In the original Unix tradition, command-line options are single letters preceded by a single hyphen. Mode-flag options that do not take following arguments can be ganged together; thus, if -a and -b are mode options, `-ab` or `-ba` is also correct and enables both. The argument to an option, if any, follows it (optionally separated by whitespace). In this style, lower-case options are preferred to upper-case. When you use upper-case options, it's good form for them to be special variants of the lower-case option.

The original Unix style evolved on slow ASR-33 teletypes that made terseness a virtue; thus the single-letter options. Holding down the shift key required actual effort; thus the preference for lower-case, and the use of "-" (rather than the perhaps more logical "+") to enable options.

The GNU style uses option keywords (rather than keyword letters) preceded by two hyphens. It evolved years later when some of the rather elaborate GNU utilities began to run out of single-letter option keys (this constituted a patch for the symptom, not a cure for the underlying disease). It remains popular because GNU options are easier to read than the alphabet soup of older styles. GNU-style options cannot be ganged together without separating whitespace. An option argument (if any) can be separated by either whitespace or a single "=" (equal-sign) character.

The GNU double-hyphen option leader was chosen so that traditional single-letter options and GNU-style keyword options could be unambiguously mixed on the same command line. Thus, if your initial design has few and simple options, you can use the Unix style without worrying about causing an incompatible 'flag day' if you need to switch to GNU style later on. On the other hand, if you are using the GNU style, it is good practice to support single-letter equivalents for at least the most common options.

The X toolkit style, confusingly, uses a single hyphen and keyword options. It is interpreted by X toolkits that filter out and process certain options (such as `-geometry` and `-display`) before handing the filtered command line to the application logic for interpretation. The X toolkit style is not properly compatible with either the classic Unix or GNU styles, and should not be used in new programs unless the value of being compatible with older X conventions seems very high.

Many tools accept a bare hyphen, not associated with any option letter, as a pseudo-filename directing the application to read from standard input. It is also conventional to recognize a double hyphen as a signal to stop option interpretation and treat all following arguments literally.

Most Unix programming languages offer libraries that will parse a command line for you in either classic-Unix or GNU style (interpreting the double-hyphen convention as well).

10.5.1 The A to Z of Command-Line Options

Over time, frequently-used options in well-known Unix programs have established a loose sort of semantic standard for what various flags might be expected to mean. The following is a list of options and meanings that should prove usefully unsurprising to an experienced Unix user:

`-a`

All (without argument). If there is a GNU-style —all option, for `-a` to be anything but a synonym for it would be quite surprising. Examples: *fuser*(1), *fetchmail*(1).

Append, as in *tar*(1). This is often paired with `-d` for delete.

`-b`

Buffer or block size (with argument). Set a critical buffer size, or (in a program having to do with archiving or managing storage media) set a block size. Examples: *du*(1), *df*(1), *tar*(1).

Batch. If the program is naturally interactive, `-b` may be used to suppress prompts or set other options appropriate to accepting input from a file rather than a human operator. Example: *flex*(1).

-c

Command (with argument). If the program is an interpreter that normally takes commands from standard input, it is expected that the option of a `-c` argument will be passed to it as a single line of input. This convention is particularly strong for shells and shell-like interpreters. Examples: *sh*(1), *ash*(1), *bsh*(1), *ksh*(1), *python*(1). Compare `-e` below.

Check (without argument). Check the correctness of the file argument(s) to the command, but don't actually perform normal processing. Frequently used as a syntax-check option by programs that do interpretation of command files. Examples: *getty*(1), *perl*(1).

-d

Debug (with or without argument). Set the level of debugging messages. This one is very common.

Occasionally `-d` has the sense of 'delete' or 'directory'.

-D

Define (with argument). Set the value of some symbol in an interpreter, compiler, or (especially) macro-processor-like application. The model is the use of `-D` by the C compiler's macro preprocessor. This is a strong association for most Unix programmers; don't try to fight it.

-e

Execute (with argument). Programs that are wrappers, or that can be used as wrappers, often allow `-e` to set the program they hand off control to. Examples: *xterm*(1), *perl*(1).

Edit. A program that can open a resource in either a read-only or editable mode may allow `-e` to specify opening in the editable mode. Examples: *crontab*(1), and the *get*(1) utility of the SCCS version-control system.

Occasionally `-e` has the sense of 'exclude' or 'expression'.

-f

File (with argument). Very often used with an argument to specify an input (or, less frequently, output) file for programs that need to randomly-access their input or output (so that redirection via < or > won't suffice). The classic example is *tar*(1); others abound. It is also used to indicate that arguments normally taken from the command line should be taken from a file instead; see *awk*(1) and *egrep*(1) for classic examples. Compare `-o` below; often, `-f` is the input-side analog of `-o`.

Force (typically without argument). Force some operation (such as a file lock or unlock) that is normally performed conditionally. This is less common.

Daemons often use `-f` in a way that combines these two meanings, to force processing of a configuration file from a non-default location. Examples: *ssh*(1), *httpd*(1), and many other daemons.

`-h`

Headers (typically without argument). Enable, suppress, or modify headers on a tabular report generated by the program. Examples: *pr*(1), *ps*(1).

Help. This is actually less common than one might expect offhand—for much of Unix's early history developers tended to think of on-line help as memory-footprint overhead they couldn't afford. Instead they wrote manual pages (this shaped the man-page style in ways we'll discuss in Chapter 18).

`-i`

Initialize (usually without argument). Set some critical resource or database associated with the program to an initial or empty state. Example: *ci*(1) in RCS.

Interactive (usually without argument). Force a program that does not normally query for confirmation to do so. There are classical examples (*rm*(1), *mv*(1)) but this use is not common.

`-I`

Include (with argument). Add a file or directory name to those searched for resources by the application. All Unix compilers with any equivalent of source-file inclusion in their languages use `-I` in this sense. It would be extremely surprising to see this option letter used in any other way.

`-k`

Keep (without argument). Suppress the normal deletion of some file, message, or resource. Examples: *passwd*(1), *bzip*(1), and *fetchmail*(1).

Occasionally `-k` has the sense of 'kill'.

`-l`

List (without argument). If the program is an archiver or interpreter/player for some kind of directory or archive format, it would be quite surprising for -l to do anything but request an item listing. Examples: *arc*(1), *binhex*(1), *unzip*(1). (However, *tar*(1) and *cpio*(1) are exceptions.)

In programs that are already report generators, `-l` almost invariably means "long" and triggers some kind of long-format display revealing more detail than the default mode. Examples: *ls*(1), *ps*(1).

Load (with argument). If the program is a linker or a language interpreter, `-l` invariably loads a library, in some appropriate sense. Examples: *gcc*(1), *f77*(1), *emacs*(1).

Login. In programs such as *rlogin*(1) and *ssh*(1) that need to specify a network identity, `-l` is how you do it.

Occasionally `-l` has the sense of 'length' or 'lock'.

`-m`

Message (with argument). Used with an argument, `-m` passes it in as a message string for some logging or announcement purpose. Examples: *ci*(1), *cvs*(1).

Occasionally `-m` has the sense of 'mail', 'mode', or 'modification-time'.

`-n`

Number (with argument). Used, for example, for page number ranges in programs such as *head*(1), *tail*(1), *nroff*(1), and *troff*(1). Some networking tools that normally display DNS names accept `-n` as an option that causes them to display the raw IP addresses instead; *ifconfig*(1) and *tcpdump*(1) are the archetypal examples.

Not (without argument). Used to suppress normal actions in programs such as *make*(1).

`-o`

Output (with argument). When a program needs to specify an output file or device by name on the command line, the `-o` option does it. Examples: *as*(1), *cc*(1), *sort*(1). On anything with a compiler-like interface, it would be extremely surprising to see this option used in any other way. Programs that support `-o` often (like gcc) have logic that allows it to be recognized after ordinary arguments as well as before.

`-p`

Port (with argument). Especially used for options that specify TCP/IP port numbers. Examples: *cvs*(1), the PostgreSQL tools, the *smbclient*(1), *snmpd*(1), *ssh*(1).

Protocol (with argument). Examples: *fetchmail*(1), *snmpnetstat*(1).

`-q`

Quiet (usually without argument). Suppress normal result or diagnostic output. This is very common. Examples: *ci*(1), *co*(1), *make*(1). See also the 'silent' sense of `-s`.

`-r` (also `-R`)

Recurse (without argument). If the program operates on a directory, then this option might tell it to recurse on all subdirectories. Any other use in a utility that operated on directories would be quite surprising. The classic example is, of course, *cp*(1).

Reverse (without argument). Examples: *ls*(1), *sort*(1). A filter might use this to reverse its normal translation action (compare `-d`).

`-s`

Silent (without argument). Suppress normal diagnostic or result output (similar to `-q`; when both are supported, q means 'quiet' but `-s` means 'utterly silent'). Examples: *csplit*(1), *ex*(1), *fetchmail*(1).

Subject (with argument). *Always* used with this meaning on commands that send or manipulate mail or news messages. It is extremely important to support this, as programs that send mail expect it. Examples: *mail*(1), *elm*(1), *mutt*(1).

Occasionally `-s` has the sense of 'size'.

`-t`

Tag (with argument). Name a location or give a string for a program to use as a retrieval key. Especially used with text editors and viewers. Examples: *cvs*(1), *ex*(1), *less*(1), *vi*(1).

`-u`

User (with argument). Specify a user, by name or numeric UID. Examples: *crontab*(1), *emacs*(1), *fetchmail*(1), *fuser*(1), *ps*(1).

`-v`

Verbose (with or without argument). Used to enable transaction-monitoring, more voluminous listings, or debugging output. Examples: *cat*(1), *cp*(1), *flex*(1), *tar*(1), many others.

Version (without argument). Display program's version on standard output and exit. Examples: *cvs*(1), *chattr*(1), *patch*(1), *uucp*(1). More usually this action is invoked by `-V`.

`-V`

Version (without argument). Display program's version on standard output and exit (often also prints compiled-in configuration details as well). Examples: *gcc*(1), *flex*(1), *hostname*(1), many others. It would be quite surprising for this switch to be used in any other way.

`-w`

Width (with argument). Especially used for specifying widths in output formats. Examples: *faces*(1), *grops*(1), *od*(1), *pr*(1), *shar*(1).

Warning (without argument). Enable warning diagnostics, or suppress them. Examples: *fetchmail*(1), *flex*(1), *nsgmls*(1).

`-x`

Enable debugging (with or without argument). Like `-d`. Examples: *sh*(1), *uucp*(1).

Extract (with argument). List files to be extracted from a archive or working set. Examples: *tar*(1), *zip*(1).

`-y`
Yes (without argument). Authorize potentially destructive actions for which the program would normally require confirmation. Examples: *fsck*(1), *rz*(1).

`-z`
Enable compression (without argument). Archiving and backup programs often use this. Examples: *bzip*(1), GNU *tar*(1), *zcat*(1), *zip*(1), *cvs*(1).

The preceding examples are taken from the Linux toolset, but should be good on most modern Unixes.

When you're choosing command-line option letters for your program, look at the manual pages for similar tools. Try to use the same option letters they use for the analogous functions of your program. Note that some particular application areas that have particularly strong conventions about command-line switches which you violate at your peril—compilers, mailers, text filters, network utilities and X software are all notable for this. Anybody who wrote a mail agent that used `-s` as anything but a Subject switch, for example, would have scorn rightly heaped upon the choice.

The GNU project recommends conventional meanings for a few double-dash options in the GNU coding standards.[5] It also lists long options which, though not standardized, are used in many GNU programs. If you are using GNU-style options, and some option you need has a function similar to one of those listed, by all means obey the Rule of Least Surprise and reuse the name.

10.5.2 Portability to Other Operating Systems

To have command-line options, you have to have a command line. The MS-DOS family does, of course, though in Windows it's hidden by a GUI and its use is discouraged; the fact that the option character is normally '`/`' rather than '`-`' is merely a detail. MacOS classic and other pure GUI environments have no close equivalent of command-line options.

10.6 How to Choose among the Methods

We've looked in turn at system and user run control files, at environment variables, and at command-line arguments. Observe the progression from least easily changed

5. See the Gnu Coding Standards (`http://www.gnu.org/prep/standards.html`).

to most easily changed. There is a strong convention that well-behaved Unix programs that use more than one of these places should look at them in the order given, allowing later settings to override earlier ones (there are specific exceptions, such as command-line options that specify where a dotfile should be found).

In particular, environment settings usually override dotfile settings, but can be overridden by command-line options. It is good practice to provide a command-line option like the `-e` of *make*(1) that can override environment settings or declarations in run-control files; that way the program can be scripted with well-defined behavior regardless of the way the run-control files look or environment variables are set.

Which of these places you choose to look at depends on how much persistent configuration state your program needs to keep around between invocations. Programs designed mainly to be used in a batch mode (as generators or filters in pipelines, for example) are usually completely configured with command-line options. Good examples of this pattern include *ls*(1), *grep*(1) and *sort*(1). At the other extreme, large programs with complicated interactive behavior may rely entirely on run-control files and environment variables, and normal use involves few command-line options or none at all. Most X window managers are a good example of this pattern.

(Unix has the capability for the same file to have multiple names or 'links'. At startup time, every program has available to it the filename through which it was called. One other way to signal to a program that has several modes of operation which one it should come up in is to give it a link for each mode, have it find out which link it was called through, and change its behavior accordingly. But this technique is generally considered unclean and seldom used.)

Let's look at a couple of programs that gather configuration data from all three places. It will be instructive to consider why, for each given piece of configuration data, it is collected as it is.

10.6.1 Case Study: *fetchmail*

The *fetchmail* program uses only two environment variables, `USER` and `HOME`. These variables are in the predefined set initialized by the system; many programs use them.

The value of `HOME` is used to find the dot file `.fetchmailrc`, which contains configuration information in a fairly elaborate syntax obeying the shell-like lexical rules described above. This is appropriate because, once it has been initially set up, Fetchmail's configuration will change only infrequently.

There is neither an `/etc/fetchmailrc` nor any other system-wide file specific to fetchmail. Normally such files hold configuration that's not specific to an individual user. fetchmail does use a small set of properties with this kind of scope—specifically, the name of the local postmaster, and a few switches and values describing the local mail transport setup (such as the port number of the local SMTP listener). In practice, however, these are seldom changed from their compiled-in default values. When they

are changed, they tend to be modified in user-specific ways. Thus, there has been no demand for a system-wide fetchmail run control file.

Fetchmail can retrieve host/login/password triples from a `.netrc` file. Thus, it gets authenticator information in the least surprising way.

Fetchmail has an elaborate set of command-line options, which nearly but do not entirely replicate what the `.fetchmailrc` can express. The set was not originally large, but grew over time as new constructs were added to the `.fetchmailrc` minilanguage and parallel command-line options for them were added more or less reflexively.

The intent of supporting all these options was to make fetchmail easier to script by allowing users to override bits of its run control from the command line. But it turns out that outside of a few options like `--fetchall` and `--verbose` there is little demand for this—and none that can't be satisfied with a shellscript that creates a temporary run-control file on the fly and then feeds it to fetchmail using the `-f` option.

Thus, most of the command-line options are never used, and in retrospect including them was probably a mistake; they bulk up the fetchmail code a bit without accomplishing anything very useful.

> If bulking up the code were the only problem, nobody would care, except for a couple of maintainers. However, options increase the chances of error in code, particularly due to unforeseen interactions among rarely used options. Worse, they bulk up the manual, which is a burden on everybody.
>
> —Doug McIlroy

There is a lesson here; had I thought carefully enough about fetchmail's usage pattern and been a little less ad-hoc about adding features, the extra complexity might have been avoided.

> An alternative way of dealing with such situations, which doesn't clutter up either the code or the manual much, is to have a "set option variable" option, such as the `-O` option of *sendmail*, which lets you specify an option name and value, and sets that name to that value as if such a setting had been given in a configuration file. A more powerful variant of this is what `ssh` does with its `-o` option: the argument to `-o` is treated as if it were a line appended to the configuration file, with the full config-file syntax available. Either of these approaches gives people with unusual requirements a way to override configuration from the command line, without requiring you to provide a separate option for each bit of configuration that might be overridden.
>
> —Henry Spencer

10.6.2 Case Study: The XFree86 Server

The X windowing system is the engine that supports bitmapped displays on Unix machines. Unix applications running through a client machine with a bitmapped display get their input events through X and send screen-painting requests to it. Confusingly, X 'servers' actually run on the client machine—they exist to serve requests to interact with the client machine's display device. The applications sending those requests to the X server are called 'X clients', even though they may be running on a server machine. And no, there is no way to explain this inverted terminology that is not confusing.

X servers have a forbiddingly complex interface to their environment. This is not surprising, as they have to deal with a wide range of complex hardware and user preferences. The environment queries common to all X servers, documented on the *X*(1) and *Xserver*(1) pages, therefore make a useful example for study. The implementation we examine here is XFree86, the X implementation used under Linux and several other open-source Unixes.

At startup, the XFree86 server examines a system-wide run-control file; the exact pathname varies between X builds on different platforms, but the basename is XF86Config. The XF86Config file has a shell-like syntax as described above. Example 10.2 is a sample section of an XF86Config file:

Example 10.2: X configuration example.

```
# The 16-color VGA server

Section "Screen"
    Driver      "vga16"
    Device      "Generic VGA"
    Monitor     "LCD Panel 1024x768"
    Subsection  "Display"
        Modes       "640x480" "800x600"
        ViewPort    0 0
    EndSubsection
EndSection
```

The XF86Config file describes the host machine's display hardware (graphics card, monitor), keyboard, and pointing device (mouse/trackball/glidepad). It's appropriate for this information to live in a system-wide run-control file, because it applies to all users of the machine.

Once X has acquired its hardware configuration from the run control file, it uses the value of the environment variable `HOME` to find two dotfiles in the calling user's home directory. These files are `.Xdefaults` and `.xinitrc`.[6]

The `.Xdefaults` file specifies per-user, application-specific resources relevant to X (trivial examples of these might include font and foreground/background colors for a terminal emulator). The phrase 'relevant to X' indicates a design problem, however. Collecting all these resource declarations in one place is convenient for inspecting and editing them, but it is not always clear what should be declared in `.Xdefaults` and what belongs in an application-specific dotfile. The `.xinitrc` file specifies the commands that should be run to initialize the user's X desktop just after server startup. These programs will almost always include a window or session manager.

X servers have a large set of command-line options. Some of these, such as the `-fp` (font path) option, override the XF86Config. Some are intended to help track server bugs, such as the `-audit` option; if these are used at all, they are likely to vary quite frequently between test runs and are therefore poor candidates to be included in a run-control file. A very important option is the one that sets the server's display number. Multiple servers may run on a host provided each has a unique display number, but all instances share the same run-control file(s); thus, the display number cannot be derived solely from those files.

10.7 On Breaking These Rules

The conventions described in this chapter are not absolute, but violating them will increase friction costs for users and developers in the future. Break them if you must—but be sure you know exactly why you are doing so before you do it. And if you do break them, make sure that attempts to do things in conventional ways break noisily, giving proper error feedback in accordance with the Rule of Repair.

6. The `.xinitrc` is analogous to a Startup folder on Windows and other operating systems.