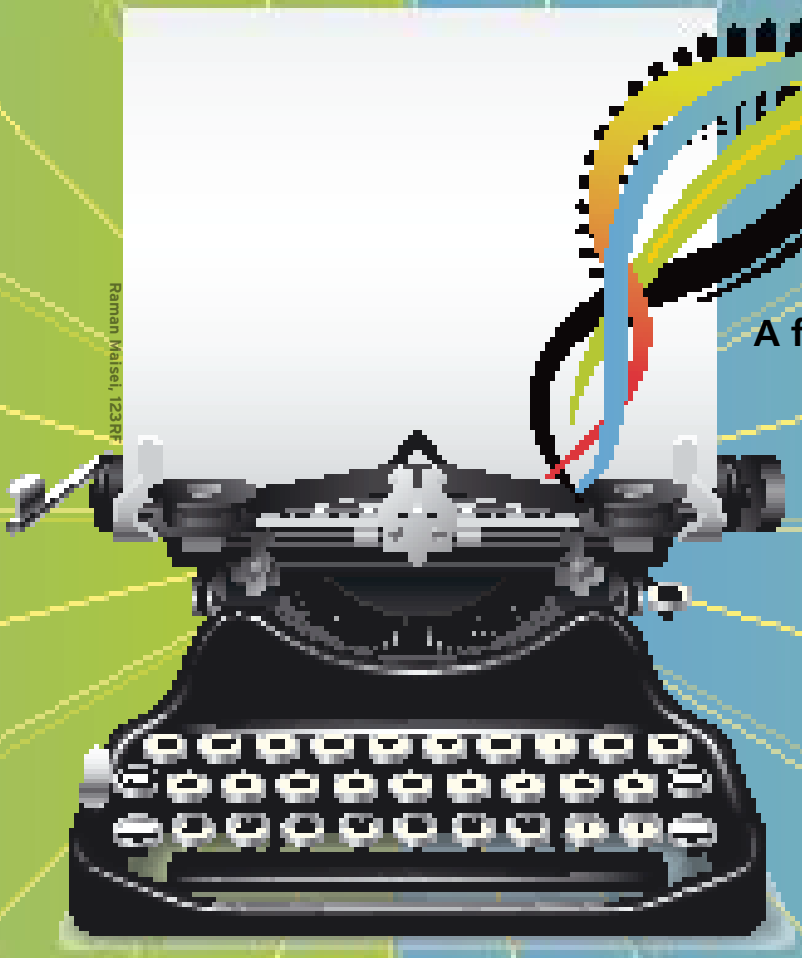

Raman Maisel, 123RF

A free history function in Perl scripts

Type Less

If you are interested in adding a neat and practical history mechanism with an editing function to an interactive Perl tool you have programmed, all you need to do is include the GNU Readline and History libraries. *By Michael Schilli*

The GNU `readline` and `history` utilities are real IT dinosaurs, but still extremely useful and widely used. They provide a mechanism for editing and repeating user input to any command-line program, and it doesn't involve much effort on the programmer's part.

Long-Winded SQL

For example, if you type a lengthy SQL query in the `mysql` MySQL client (Figure 1), you will definitely appreciate the ability to press the Up arrow key to repeat the input at the next prompt. This way, you can either repeat the command, or modify it and resubmit it afterward. Incidentally, these functions don't rely on the MySQL client's `\e` command, which opens a full-blown editor to make modifications. Instead, if you enter a long command line and then discover one of the words contains a typo, you can simply press the Left arrow key to go back to the start of the line and correct the command before you submit it.

As a quick inspection of the MySQL source code reveals, the database doesn't implement this practical mechanism itself; instead, it uses the C `readline()` and `add_history()` functions from the GNU Readline and History libraries to read user input and add the selected commands to a pool for access at some later time. As with all GNU tools, invoking the documentation tool with `info readline` or `info history` shows the man pages for the utilities in a genuine 1970s look. Quickly, a few browsing tips: Pressing the `N` key tells `info` to jump to the next chapter, whereas `P`

LISTING 1: readline-test

```
01 #!/usr/local/bin/perl -w
02 use strict;
03
04 use Term::ReadLine;
05 my $term =
06   Term::ReadLine->new(
07   "myapp");
08
09 while (1) {
10
11   my $input =
12     $term->readline(
13     "input>");
14
15   last unless defined $input;
16   print "Input was '$input'\n";
17
18   if ( $input =~ /\S/ ) {
19     $term->addhistory(
20       $input);
21   }
22 }
```

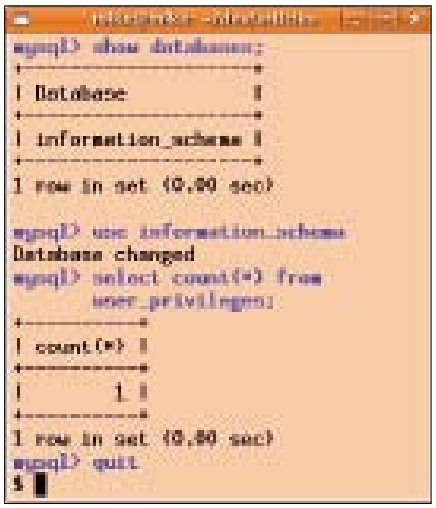

Figure 1: A command-line session with the `mysql` MySQL client.

takes you back to the last chapter you viewed, and the Tab key moves to the next linked bullet on a page.

For the last session, you can still access the history after restarting the MySQL client. How does this work? As Figure 2 shows, the GNU history mechanism used in this example dumps the information into the `~/.mysql_history` file. The final command, `quit`, doesn't appear in the history file because MySQL only saves useful commands and notices that there is nothing worthwhile to save. The program exits before the call to `add_history()` occurs.

Spoiled for Perl

Perl offers spoiled script programmers a convenient interface to the GNU libraries. The Perl Term::ReadLine::Gnu module from CPAN communicates with the C layer of the installed GNU libraries and offers the Perl programmer an object-oriented layer. The Term::ReadLine module is included with Perl distributions by default, although it only offers restricted functionality. For a fully functional

LISTING 2: wrapper-test

```
01 #!/usr/local/bin/perl -w
02 use strict;
03
04 $| = 1;
05 for ( 1 .. 3 ) {
06   print "Input> ";
07   my $in = <STDIN>;
08   chomp $in;
09   print "You said '$in'\n";
10 }
```

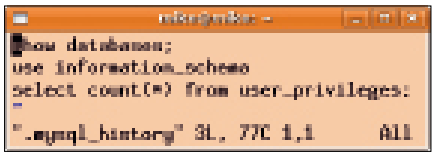

Figure 2: Command line input is stored in the `~/.mysql_history` file for later sessions.

Term::ReadLine, you need to install Term::ReadLine::Gnu from CPAN.

Listing 1 creates a Term::ReadLine object and calls its `readline()` method, which in turn prompts the user to type a command. If it contains usable characters, it makes sense to call `add_history()` to add it to the line buffer and be able to dig it out again later with the use of arrow keys.

The sample listing takes the easy way out here and accepts anything apart from all-blank lines as useful input, but of course, I could imagine some sort of sophisticated input validation instead. For more information on terminal programming in Perl, refer to the man pages of the two aforementioned CPAN modules, or some fairly spartan documentation scattered throughout several Perl books [2].

Garbled Characters in the Debugger

The internal Perl debugger also has a history mechanism that avoids users having to continually type the same old commands. But on certain installations, if you press the arrow keys to pull up previous commands, all you get is garbled characters, such as:

```
perl -d test.pl
DB<1> $ ^[[A,
```

This kind of output is a sure sign that the victim has forgotten to install the Perl wrapper for the GNU Readline library from CPAN as in:

```
cpan> install Term::ReadLine::Gnu
```

What's going on? When launched, Perl's debugger checks to see whether the `Gnu` module really is available and, if not, provides a functional but restricted terminal environment without a history function.

Without some manual attention, cursor navigation with Gnu Readline uses Emacs commands, which might sound strange for fans of Vi. Rumor has it that there are people out there who actually broke their fingers typing complicated Emacs keyboard shortcuts. The following option

```
set editing-mode vi
```

in `~/.inputrc` below your home directory will save you from this fate by automatically shifting Readline into Vi mode.

If you don't notice that Readline is in the wrong editor mode until you have started typing, Meta+Ctrl+J switches modes. This looks very much like an Emacs-only command, but Vi mode understands it, too, and switches to Emacs mode. If your keyboard doesn't have a Meta key, simply tap the Esc key and then press Ctrl+J.

Instead of pressing Ctrl+B to move the cursor to the left, Vi aficionados would then press Esc to switch to command mode and then press `H` until the cursor reached the desired position. Pressing `I` takes you back to insert mode.

Writing History

In a history of dozens of commands, users will find what they are looking for more quickly if they search for certain

LISTING 3: readline-complete

```
01 #!/usr/local/bin/perl -w
02 use strict;
03
04 use Term::ReadLine;
05 my $term =
06   Term::ReadLine->new(
07   'myapp');
08 my $attribs = $term->Attribs;
09 $attribs
10   ->{completion_entry_function}
11   = $attribs
12   ->{list_completion_function};
13
14 $attribs->{completion_word} =
15   [qw(install remove quit)];
16
17 while (1) {
18   my $cmd = $term->readline(
19     "myapp> ");
20   last if $cmd =~ /^quit/i;
21 }
```

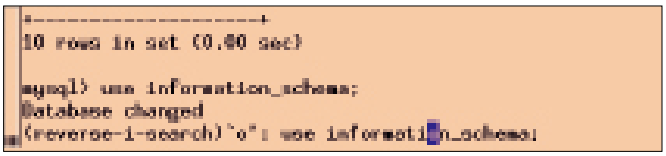

Figure 3: In Emacs search mode, Readline fetches the last command containing an "o" when you press the o key …

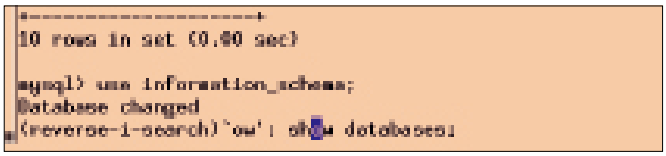

Figure 4: … and if you type ow, the mechanism will narrow down the result list to commands containing the latter.

entries instead of just browsing through pages full of inappropriate commands. In Vi mode, the Esc key takes you back to command mode, where you can then type a slash, followed by parts of the search string you are looking for and then the Return key to view a list of matches. To scroll through the result list, press N (next) and P (previous).

Once you have the history entry you were looking for, the Return key executes it, but you can use familiar Vi commands to edit the command line. In Emacs mode, Ctrl+R searches backward and displays matches for the string you entered in this active search mode (Figures 3 and 4).

Forced Development

Programs coded without a Readline function, and which thus do have the ability to remember and edit input lines, can be taught the required tricks with the rlwrap [3] wrapper. Listing 2 shows a simple Perl script that accepts input from the command line three times, using the typical Perl <STDIN> construct, and then outputs the user input.

In Figure 5, you can see how the user presses the Up arrow key to access the last entry, but instead harvests a garbled line, ^[[A. In contrast, the user launches the same script in Figure 6 but with the rlwrap wrapper, and hey presto, pressing the Up arrow key in line three conjures the data typed in the second line of input back onto the screen. Because of persistent storage, the data will even be available on a subsequent run of the script.

A close look at your home directory shows a .wrapper-test_history file.

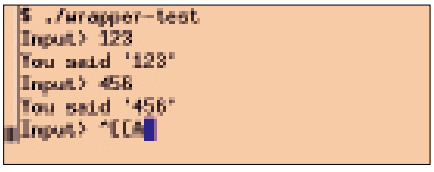

Figure 5: Without Readline support, the arrow key only generates garbled output rather than finding previous entries.

How does this work? The wrapper rlwrap simply uses LD_PRELOAD to overload the input functions of the original program and replaces them with wrappers that collaborate with GNU's Readline and History libraries.

You Complete Me

Readline not only offers an editing function but will finish off incomplete command lines when you press the Tab key. Just like the Bash completion mechanism, which I discussed in a previous column [4], users can customize this function.

Listing 2 gives an example of a simple command interpreter that only understands the install, remove, and quit commands. The API for the command extension to the Readline library is somewhat complex; the Readline object's completion_entry_function entry expects a callback, which Readline will call multiple times if the user presses Tab once, until all the suggestions are made available.

Readline adds two parameters whenever the callback runs: $count and $word. The $word parameter is the word to be completed – that is, the string at the end of which the cursor was located when the user pressed Tab. For the first call, $count is 0 and then is incremented for all following calls.

In other words, the callback function is expected to initialize itself and return the first possible completion when $count is equal to 0 and to return the next option from a list of possible completions while $count is increasing on subsequent calls. A return value of undef by the callback indicates to Readline that the end of the list has been reached and all possible completions have been exhausted.

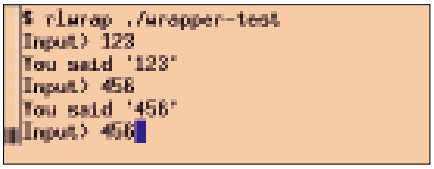

Figure 6: The rlwrap wrapper adds Readline support without modifying the program and thus conjures up the previous command lines on request.

If only one completion exists for the word, the callback function returns the results when called, with $count equal to 0 and undef for $count equal to 1. Fortunately, for programmers, Term::ReadLine::Gnu already has a callback function for simple applications; it is available in the $attribs reference of the list_completion_function. This function will complete any words it finds in a special array below the completion_word hash entry.

Instead of having to write a callback function, programmers can simply store a reference to an array of keywords possible to complete in the completion_word hash entry and then set the value of the completion_entry_function entry to the function reference of list_completion_function. This covers handling multiple calls with different counts automatically.

Now if the script readline-complete in Listing 3 asks for a command after showing the myapp> prompt, and if the user then presses I+Tab, Readline will thus complete the user's input to the install command, the only possible command that begins with that letter. It will save the user six key presses in the process, which doesn't sound like much but tends to add up over the course of a programmer's work day, and will go easy on those never-resting fingers and their stressed tendons. ■■■

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Resources/Article-Code*

[2] Wainwright, Peter. *Pro Perl.* Apress Verlag, 2005, pg. 551

[3] *rlwrap* wrapper : *http://utopia.knoware.nl/~hlub/rlwrap/man.html*

[4] "Perl Completion" by Michael Schilli, *Linux Magazine*, May 2010, pg. 71