# Linenoise

A minimal, zero-config, BSD licensed, readline replacement used in Redis, MongoDB, and Android.

* Single and multi line editing mode with the usual key bindings implemented.
* History handling.
* Completion.
* Hints (suggestions at the right of the prompt as you type).
* About 1,100 lines of BSD license source code.
* Only uses a subset of VT100 escapes (ANSI.SYS compatible).

## Can a line editing library be 20k lines of code?

Line editing with some support for history is a really important feature for command line utilities. Instead of retyping almost the same stuff again and again it's just much better to hit the up arrow and edit on syntax errors, or in order to try a slightly different command. But apparently code dealing with terminals is some sort of Black Magic: readline is 30k lines of code, libedit 20k. Is it reasonable to link small utilities to huge libraries just to get a minimal support for line editing?

So what usually happens is either:

* Large programs with configure scripts disabling line editing if readline is not present in the system, or not supporting it at all since readline is GPL licensed and libedit (the BSD clone) is not as known and available as readline is (Real world example of this problem: Tclsh).
* Smaller programs not using a configure script not supporting line editing at all (A problem we had with Redis-cli for instance).

The result is a pollution of binaries without line editing support.

So I spent more or less two hours doing a reality check resulting in this little library: is it *really* needed for a line editing library to be 20k lines of code? Apparently not, it is possibe to get a very small, zero configuration, trivial to embed library, that solves the problem. Smaller programs will just include this, supporing line editing out of the box. Larger programs may use this little library or just checking with configure if readline/libedit is available and resorting to Linenoise if not.

## Terminals, in 2010.

Apparently almost every terminal you can happen to use today has some kind of support for basic VT100 escape sequences. So I tried to write a lib using just very basic VT100 features. The resulting library appears to work everywhere I tried to use it, and now can work even on ANSI.SYS compatible terminals, since no VT220 specific sequences are used anymore.

The library is currently about 1100 lines of code. In order to use it in your project just look at the *example.c* file in the source distribution, it is trivial. Linenoise is BSD code, so you can use both in free software and commercial software.

## Tested with...

* Linux text only console ($TERM = linux)
* Linux KDE terminal application ($TERM = xterm)
* Linux xterm ($TERM = xterm)
* Linux Buildroot ($TERM = vt100)
* Mac OS X iTerm ($TERM = xterm)
* Mac OS X default Terminal.app ($TERM = xterm)
* OpenBSD 4.5 through an OSX Terminal.app ($TERM = screen)
* IBM AIX 6.1
* FreeBSD xterm ($TERM = xterm)
* ANSI.SYS
* Emacs comint mode ($TERM = dumb)

Please test it everywhere you can and report back!

## Let's push this forward!

Patches should be provided in the respect of Linenoise sensibility for small easy to understand code.

Send feedbacks to antirez at gmail

# The API

Linenoise is very easy to use, and reading the example shipped with the library should get you up to speed ASAP. Here is a list of API calls and how to use them.

char \*linenoise(const char \*prompt);



This is the main Linenoise call: it shows the user a prompt with line editing and history capabilities. The prompt you specify is used as a prompt, that is, it will be printed to the left of the cursor. The library returns a buffer with the line composed by the user, or NULL on end of file or when there is an out of memory condition.

When a tty is detected (the user is actually typing into a terminal session) the maximum editable line length is LINENOISE\_MAX\_LINE. When instead the standard input is not a tty, which happens every time you redirect a file to a program, or use it in an Unix pipeline, there are no limits to the length of the line that can be returned.

The returned line should be freed with the free() standard system call. However sometimes it could happen that your program uses a different dynamic allocation library, so you may also used linenoiseFree to make sure the line is freed with the same allocator it was created.

The canonical loop used by a program using Linenoise will be something like this:

while((line = linenoise("hello> ")) != NULL) {

printf("You wrote: %s\n", line);

linenoiseFree(line); /\* Or just free(line) if you use libc malloc. \*/

}

## Single line VS multi line editing

By default, Linenoise uses single line editing, that is, a single row on the screen will be used, and as the user types more, the text will scroll towards left to make room. This works if your program is one where the user is unlikely to write a lot of text, otherwise multi line editing, where multiple screens rows are used, can be a lot more comfortable.

In order to enable multi line editing use the following API call:

linenoiseSetMultiLine(1);



You can disable it using 0 as argument.

## History

Linenoise supporst history, so that the user does not have to retype again and again the same things, but can use the down and up arrows in order to search and re-edit already inserted lines of text.

The followings are the history API calls:

int linenoiseHistoryAdd(const char \*line);

int linenoiseHistorySetMaxLen(int len);

int linenoiseHistorySave(const char \*filename);

int linenoiseHistoryLoad(const char \*filename);



Use linenoiseHistoryAdd every time you want to add a new element to the top of the history (it will be the first the user will see when using the up arrow).

Note that for history to work, you have to set a length for the history (which is zero by default, so history will be disabled if you don't set a proper one). This is accomplished using the linenoiseHistorySetMaxLen function.

Linenoise has direct support for persisting the history into an history file. The functions linenoiseHistorySave and linenoiseHistoryLoad do just that. Both functions return -1 on error and 0 on success.

## Completion

Linenoise supports completion, which is the ability to complete the user input when she or he presses the <TAB> key.

In order to use completion, you need to register a completion callback, which is called every time the user presses <TAB>. Your callback will return a list of items that are completions for the current string.

The following is an example of registering a completion callback:

linenoiseSetCompletionCallback(completion);



The completion must be a function returning void and getting as input a const char pointer, which is the line the user has typed so far, and a linenoiseCompletions object pointer, which is used as argument of linenoiseAddCompletion in order to add completions inside the callback. An example will make it more clear:

void completion(const char \*buf, linenoiseCompletions \*lc) {

if (buf[0] == 'h') {

linenoiseAddCompletion(lc,"hello");

linenoiseAddCompletion(lc,"hello there");

}

}



Basically in your completion callback, you inspect the input, and return a list of items that are good completions by using linenoiseAddCompletion.

If you want to test the completion feature, compile the example program with make, run it, type h and press <TAB>.

## Hints

Linenoise has a feature called *hints* which is very useful when you use Linenoise in order to implement a REPL (Read Eval Print Loop) for a program that accepts commands and arguments, but may also be useful in other conditions.

The feature shows, on the right of the cursor, as the user types, hints that may be useful. The hints can be displayed using a different color compared to the color the user is typing, and can also be bold.

For example as the user starts to type "git remote add", with hints it's possible to show on the right of the prompt a string <name> <url>.

The feature works similarly to the history feature, using a callback. To register the callback we use:

linenoiseSetHintsCallback(hints);



The callback itself is implemented like this:

char \*hints(const char \*buf, int \*color, int \*bold) {

if (!strcasecmp(buf,"git remote add")) {

\*color = 35;

\*bold = 0;

return " <name> <url>";

}

return NULL;

}



The callback function returns the string that should be displayed or NULL if no hint is available for the text the user currently typed. The returned string will be trimmed as needed depending on the number of columns available on the screen.

It is possible to return a string allocated in dynamic way, by also registering a function to deallocate the hint string once used:

void linenoiseSetFreeHintsCallback(linenoiseFreeHintsCallback \*);



The free hint callback will just receive the pointer and free the string as needed (depending on how the hits callback allocated it).

As you can see in the example above, a color (in xterm color terminal codes) can be provided together with a bold attribute. If no color is set, the current terminal foreground color is used. If no bold attribute is set, non-bold text is printed.

Color codes are:

red = 31

green = 32

yellow = 33

blue = 34

magenta = 35

cyan = 36

white = 37;

## Screen handling

Sometimes you may want to clear the screen as a result of something the user typed. You can do this by calling the following function:

void linenoiseClearScreen(void);

