

Readline Ninja Skills

Jack Rosenthal

2016-03-07

2018-03-08

Mines Linux Users Group

<https://lug.mines.edu>

- A library for interactive line editing that your shell probably uses.
- Responsible for things like tab completion, history expansion, and all of those useful keystrokes
- Readline saves you keystrokes.
- Some readline things can make you look like a total ninja.
- Some readline things make you feel like a total ninja.

- A library for interactive line editing that your shell probably uses.
- Responsible for things like tab completion, history expansion, and all of those useful keystrokes
- Readline saves you keystrokes.
- Some readline things can make you look like a total ninja.
- Some readline things make you feel like a total ninja.

- A library for interactive line editing that your shell probably uses.
- Responsible for things like tab completion, history expansion, and all of those useful keystrokes
- Readline saves you keystrokes.
- Some readline things can make you look like a total ninja.
- Some readline things make you feel like a total ninja.

- A library for interactive line editing that your shell probably uses.
- Responsible for things like tab completion, history expansion, and all of those useful keystrokes
- Readline saves you keystrokes.
- Some readline things can make you look like a total ninja.
- Some readline things make you feel like a total ninja.

- A library for interactive line editing that your shell probably uses.
- Responsible for things like tab completion, history expansion, and all of those useful keystrokes
- Readline saves you keystrokes.
- Some readline things can make you look like a total ninja.
- Some readline things make you feel like a total ninja.

Using Readline & History

History

Readline can track your history, most shells let you use the history builtin to view your history.

You can navigate your history using the up and down keys.

Tab completion

Most of us already know what this and would die without it.

Event Designators

- **!** - begin history expansion
- **!!** - refer to the last command
- **!*n*** - refer to the *n*-th command in history
- **!-*n*** - refer to the current command minus *n*
- **!#** - refer to the current command you are typing
- **!*search*** - refer to the last command that starts with *search*
- **!?*search*?** - refer to the last command with *search* anywhere in the command

Examples:

- **sudo !!** - run the last command with `sudo` in front
- **!ls** - run the last command you typed beginning with `ls`

Event Designators

- **!** - begin history expansion
- **!!** - refer to the last command
- **!*n*** - refer to the *n*-th command in history
- **!-*n*** - refer to the current command minus *n*
- **!#** - refer to the current command you are typing
- **!*search*** - refer to the last command that starts with *search*
- **!?*search*?** - refer to the last command with *search* anywhere in the command

Examples:

- **sudo !!** - run the last command with **sudo** in front
- **!ls** - run the last command you typed beginning with **ls**

Event Designators

- **!** - begin history expansion
- **!!** - refer to the last command
- **!*n*** - refer to the *n*-th command in history
- **!-*n*** - refer to the current command minus *n*
- **!#** - refer to the current command you are typing
- **!*search*** - refer to the last command that starts with *search*
- **!?*search*?** - refer to the last command with *search* anywhere in the command

Examples:

- **sudo !!** - run the last command with sudo in front
- **!p** - run the last command you typed beginning with p

Event Designators

- `!` - begin history expansion
- `!!` - refer to the last command
- `!n` - refer to the *n*-th command in history
- `!-n` - refer to the current command minus *n*
- `!#` - refer to the current command you are typing
- `!search` - refer to the last command that starts with *search*
- `!?search?` - refer to the last command with *search* anywhere in the command

Examples:

- `sudo !!` - run the last command with `sudo` in front
- `!ls` - run the last command you typed beginning with `ls`

Event Designators

- `!` - begin history expansion
- `!!` - refer to the last command
- `!n` - refer to the *n*-th command in history
- `!-n` - refer to the current command minus *n*
- `!#` - refer to the current command you are typing
- `!search` - refer to the last command that starts with *search*
- `!?search?` - refer to the last command with *search* anywhere in the command

Examples:

- `!sudo` - get the last command which starts in front of the word sudo
- `!#` - get the last command you typed beginning with `!`

Event Designators

- `!` - begin history expansion
- `!!` - refer to the last command
- `!n` - refer to the *n*-th command in history
- `!-n` - refer to the current command minus *n*
- `!#` - refer to the current command you are typing
- `!search` - refer to the last command that starts with *search*
`!?search?` - refer to the last command with *search* anywhere in the command

Examples:

- `sudo !!` - run the last command with `sudo` in front
- `sudo !#` - run the current command with `sudo` in front

Event Designators

- `!` - begin history expansion
- `!!` - refer to the last command
- `!n` - refer to the *n*-th command in history
- `!-n` - refer to the current command minus *n*
- `!#` - refer to the current command you are typing
- `!search` - refer to the last command that starts with *search*
 `!?search?` - refer to the last command with *search* anywhere in the command

Examples:

- `sudo !!` - run the last command with `sudo` in front
- `!grep` - run the last command you typed beginning with `grep`

Event Designators

- `!` - begin history expansion
- `!!` - refer to the last command
- `!n` - refer to the *n*-th command in history
- `!-n` - refer to the current command minus *n*
- `!#` - refer to the current command you are typing
- `!search` - refer to the last command that starts with *search*
- `!?search?` - refer to the last command with *search* anywhere in the command

Examples:

- `sudo !!` - run the last command with sudo in front
- `!grep` - run the last command you typed beginning with grep

Event Designators

- `!` - begin history expansion
- `!!` - refer to the last command
- `!n` - refer to the *n*-th command in history
- `!-n` - refer to the current command minus *n*
- `!#` - refer to the current command you are typing
- `!search` - refer to the last command that starts with *search*
- `!search?` - refer to the last command with *search* anywhere in the command

Examples:

- `sudo !!` - run the last command with sudo in front
- `!grep` - run the last command you typed beginning with grep

Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (:) and then a word designator.

- `$n` - select argument `n` (zero indexed)
- `$*` - select arguments 0 through `n`
- `$@` - select the last argument (think of a regex)
- `$0` - select the command name (equivalent to `$1`)
- `$?` - select the argument that matches `$search`

Examples:

- `$1` - select the first argument of the last command
- `$2` - select the file that is the last argument of two commands

Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (:) and then a word designator.

- `%n` - select argument `n` (zero indexed)

- `%n:m` - select arguments `n` through `m`.

- `%*` - select all arguments, start of a range.

- `%?` - select the argument matching the word designator (usually `%*`).

- `%P` - select the argument that matches `%?`.

Examples:

- `%1` - select the first argument (the log command)

- `%2` - select the second argument (the log command)

Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (:) and then a word designator.

- `:n` - select argument n (zero indexed)
- `:n-m` - select arguments n through m
- `:$` - select the last argument (think of a regex)
- `:*` - select all arguments, omitting the command name (equivalent to `:1-$`)
- `:%` - select the argument that matches *?search?*

Examples:

- `echo 1 2 3` - prints the first argument of the last command
- `echo 1 2 3; echo 4 5 6` - prints the last argument of the last command

Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (:) and then a word designator.

- `:n` - select argument *n* (zero indexed)
- `:n-m` - select arguments *n* through *m*
- `:$` - select the last argument (think of a regex)
- `:*` - select all arguments, omitting the command name (equivalent to `:1-$`)
- `:%` - select the argument that matches *?search?*

Examples:

- `echo $0` - prints the command name
- `echo $1` - prints the first argument
- `echo $2` - prints the second argument
- `echo $3` - prints the third argument
- `echo $4` - prints the fourth argument
- `echo $5` - prints the fifth argument
- `echo $6` - prints the sixth argument
- `echo $7` - prints the seventh argument
- `echo $8` - prints the eighth argument
- `echo $9` - prints the ninth argument
- `echo $10` - prints the tenth argument
- `echo $11` - prints the eleventh argument
- `echo $12` - prints the twelfth argument
- `echo $13` - prints the thirteenth argument
- `echo $14` - prints the fourteenth argument
- `echo $15` - prints the fifteenth argument
- `echo $16` - prints the sixteenth argument
- `echo $17` - prints the seventeenth argument
- `echo $18` - prints the eighteenth argument
- `echo $19` - prints the nineteenth argument
- `echo $20` - prints the twentieth argument
- `echo $21` - prints the twenty-first argument
- `echo $22` - prints the twenty-second argument
- `echo $23` - prints the twenty-third argument
- `echo $24` - prints the twenty-fourth argument
- `echo $25` - prints the twenty-fifth argument
- `echo $26` - prints the twenty-sixth argument
- `echo $27` - prints the twenty-seventh argument
- `echo $28` - prints the twenty-eighth argument
- `echo $29` - prints the twenty-ninth argument
- `echo $30` - prints the thirtieth argument
- `echo $31` - prints the thirty-first argument
- `echo $32` - prints the thirty-second argument
- `echo $33` - prints the thirty-third argument
- `echo $34` - prints the thirty-fourth argument
- `echo $35` - prints the thirty-fifth argument
- `echo $36` - prints the thirty-sixth argument
- `echo $37` - prints the thirty-seventh argument
- `echo $38` - prints the thirty-eighth argument
- `echo $39` - prints the thirty-ninth argument
- `echo $40` - prints the fortieth argument
- `echo $41` - prints the forty-first argument
- `echo $42` - prints the forty-second argument
- `echo $43` - prints the forty-third argument
- `echo $44` - prints the forty-fourth argument
- `echo $45` - prints the forty-fifth argument
- `echo $46` - prints the forty-sixth argument
- `echo $47` - prints the forty-seventh argument
- `echo $48` - prints the forty-eighth argument
- `echo $49` - prints the forty-ninth argument
- `echo $50` - prints the fiftieth argument
- `echo $51` - prints the fifty-first argument
- `echo $52` - prints the fifty-second argument
- `echo $53` - prints the fifty-third argument
- `echo $54` - prints the fifty-fourth argument
- `echo $55` - prints the fifty-fifth argument
- `echo $56` - prints the fifty-sixth argument
- `echo $57` - prints the fifty-seventh argument
- `echo $58` - prints the fifty-eighth argument
- `echo $59` - prints the fifty-ninth argument
- `echo $60` - prints the sixtieth argument
- `echo $61` - prints the sixty-first argument
- `echo $62` - prints the sixty-second argument
- `echo $63` - prints the sixty-third argument
- `echo $64` - prints the sixty-fourth argument
- `echo $65` - prints the sixty-fifth argument
- `echo $66` - prints the sixty-sixth argument
- `echo $67` - prints the sixty-seventh argument
- `echo $68` - prints the sixty-eighth argument
- `echo $69` - prints the sixty-ninth argument
- `echo $70` - prints the seventieth argument
- `echo $71` - prints the seventy-first argument
- `echo $72` - prints the seventy-second argument
- `echo $73` - prints the seventy-third argument
- `echo $74` - prints the seventy-fourth argument
- `echo $75` - prints the seventy-fifth argument
- `echo $76` - prints the seventy-sixth argument
- `echo $77` - prints the seventy-seventh argument
- `echo $78` - prints the seventy-eighth argument
- `echo $79` - prints the seventy-ninth argument
- `echo $80` - prints the eightieth argument
- `echo $81` - prints the eighty-first argument
- `echo $82` - prints the eighty-second argument
- `echo $83` - prints the eighty-third argument
- `echo $84` - prints the eighty-fourth argument
- `echo $85` - prints the eighty-fifth argument
- `echo $86` - prints the eighty-sixth argument
- `echo $87` - prints the eighty-seventh argument
- `echo $88` - prints the eighty-eighth argument
- `echo $89` - prints the eighty-ninth argument
- `echo $90` - prints the ninetieth argument
- `echo $91` - prints the ninety-first argument
- `echo $92` - prints the ninety-second argument
- `echo $93` - prints the ninety-third argument
- `echo $94` - prints the ninety-fourth argument
- `echo $95` - prints the ninety-fifth argument
- `echo $96` - prints the ninety-sixth argument
- `echo $97` - prints the ninety-seventh argument
- `echo $98` - prints the ninety-eighth argument
- `echo $99` - prints the ninety-ninth argument
- `echo $100` - prints the hundredth argument

Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (:) and then a word designator.

- `:n` - select argument *n* (zero indexed)
- `:n-m` - select arguments *n* through *m*
- `:$` - select the last argument (think of a regex)
- `:*` - select all arguments, omitting the command name (equivalent to `:1-$`)
- `:%` - select the argument that matches *?search?*

Examples:

- `ls -l | grep ^d | wc -l` - count the number of directories
- `ls -l | grep ^d | wc -l | wc -l` - count the number of lines of the previous command

Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (:) and then a word designator.

- `:n` - select argument *n* (zero indexed)
- `:n-m` - select arguments *n* through *m*
- `:$` - select the last argument (think of a regex)
- `:*` - select all arguments, omitting the command name (equivalent to `:1-$`)
- `:%` - select the argument that matches *?search?*

Examples:

- `cd 1111` - cd to the first argument of the last command.
- `cd 1111 2222` - cd to the second argument of the last command.

Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (:) and then a word designator.

- `:n` - select argument *n* (zero indexed)
- `:n-m` - select arguments *n* through *m*
- `:$` - select the last argument (think of a regex)
- `:*` - select all arguments, omitting the command name (equivalent to `:1-$`)
- `:%` - select the argument that matches *?search?*

Examples:

■ `cd 1-1` - cd to the first argument of the last command

■ `vi 1-2:$` - edit the file that is the last argument of two commands ago

Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (:) and then a word designator.

- `:n` - select argument *n* (zero indexed)
- `:n-m` - select arguments *n* through *m*
- `:$` - select the last argument (think of a regex)
- `:*` - select all arguments, omitting the command name (equivalent to `:1-$`)
- `:%` - select the argument that matches *?search?*

Examples:

- `cd !:1` - cd to the first argument of the last command.
- `vim !-2:$` - edit the file that is the last argument of two commands ago

Word Designators

Often times you will want only part of a command, so you can use word designators to select which parts you want. Follow an event designator with a colon (:) and then a word designator.

- `:n` - select argument *n* (zero indexed)
- `:n-m` - select arguments *n* through *m*
- `:$` - select the last argument (think of a regex)
- `:*` - select all arguments, omitting the command name (equivalent to `:1-$`)
- `:%` - select the argument that matches *?search?*

Examples:

- `cd !!:1` - cd to the first argument of the last command.
- `vim !-2:$` - edit the file that is the last argument of two commands ago

Modifiers

Modifiers let you chop up the history expansion in ways that you like. You can chain any amount of modifiers that you would like onto your expansion.

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/search/replace/` - sed style substitution
- `:gs/search/replace/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

Modifiers

Modifiers let you chop up the history expansion in ways that you like. You can chain any amount of modifiers that you would like onto your expansion.

- **:r** - Chop off the extension of a filename
- :h - Remove the filename component, leaving only the directory (think of head)
- :t - Remove the directory component, leaving only the filename (think of tail)
- :q - Quote each of the arguments
- :s/*search/replace/* - sed style substitution
- :gs/*search/replace/* - sed style substitution, globally
- :p - print the history expansion, don't execute quite yet

Modifiers

Modifiers let you chop up the history expansion in ways that you like. You can chain any amount of modifiers that you would like onto your expansion.

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/search/replace/` - sed style substitution
- `:gs/search/replace/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

Modifiers

Modifiers let you chop up the history expansion in ways that you like. You can chain any amount of modifiers that you would like onto your expansion.

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/search/replace/` - sed style substitution
- `:gs/search/replace/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

Modifiers

Modifiers let you chop up the history expansion in ways that you like. You can chain any amount of modifiers that you would like onto your expansion.

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/search/replace/` - sed style substitution
- `:gs/search/replace/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

Modifiers

Modifiers let you chop up the history expansion in ways that you like. You can chain any amount of modifiers that you would like onto your expansion.

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/search/replace/` - sed style substitution
- `:gs/search/replace/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

Modifiers

Modifiers let you chop up the history expansion in ways that you like. You can chain any amount of modifiers that you would like onto your expansion.

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/search/replace/` - sed style substitution
- `:gs/search/replace/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

Modifiers

Modifiers let you chop up the history expansion in ways that you like. You can chain any amount of modifiers that you would like onto your expansion.

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/search/replace/` - sed style substitution
- `:gs/search/replace/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

Modifiers

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/search/replace/` - sed style substitution
- `:gs/search/replace/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

Examples:

```
■ mv important.png !#:1:r.gif - rename important.png to important.gif
■ touch mydir/file.txt
■ cd !$:h
```

Modifiers

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/search/replace/` - sed style substitution
- `:gs/search/replace/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

Examples:

- `mv important.png !#:1:r.gif` - rename important.png to important.gif
- `touch mydir/file.txt`
- `cd !$:h`

Modifiers

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/search/replace/` - sed style substitution
- `:gs/search/replace/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

Examples:

- `mv important.png !#:1:r.gif` - rename important.png to important.gif
- `touch mydir/file.txt`
- `cd !$:h`

Modifiers

- `:r` - Chop off the extension of a filename
- `:h` - Remove the filename component, leaving only the directory (think of head)
- `:t` - Remove the directory component, leaving only the filename (think of tail)
- `:q` - Quote each of the arguments
- `:s/search/replace/` - sed style substitution
- `:gs/search/replace/` - sed style substitution, globally
- `:p` - print the history expansion, don't execute quite yet

Examples:

- `mv important.png !#:1:r.gif` - rename important.png to important.gif
- `touch mydir/file.txt`
- `cd !$:h`

Abbreviations Allowed

- `!!:...` can be shortened to `!:...`
- The `:` can be removed from word designators where it is unambiguous. So `!$` and `!*` are allowed.
- The trailing `/` in a substitution can be omitted if it is unambiguous that the substitution has ended.
- The trailing `?` in a `!?search?` can be omitted for the same reason.
- Any delimiter can be used in a substitution, so `!:s x find x replace x` is legal.

Abbreviations Allowed

- `!!:...` can be shortened to `!:...`
- The `:` can be removed from word designators where it is unambiguous. So `!$` and `!*` are allowed.
- The trailing `/` in a substitution can be omitted if it is unambiguous that the substitution has ended.
- The trailing `?` in a `!?search?` can be omitted for the same reason.
- Any delimiter can be used in a substitution, so `!:sxfindxreplacx` is legal.

Abbreviations Allowed

- `!!:...` can be shortened to `!:...`
- The `:` can be removed from word designators where it is unambiguous. So `!$` and `!*` are allowed.
- The trailing `/` in a substitution can be omitted if it is unambiguous that the substitution has ended.
- The trailing `?` in a `!?search?` can be omitted for the same reason.
- Any delimiter can be used in a substitution, so `!:sxfindxreplacex` is legal.

Abbreviations Allowed

- `!!:...` can be shortened to `!:...`
- The `:` can be removed from word designators where it is unambiguous. So `!$` and `!*` are allowed.
- The trailing `/` in a substitution can be omitted if it is unambiguous that the substitution has ended.
- The trailing `?` in a `!?search?` can be omitted for the same reason.
- Any delimiter can be used in a substitution, so `! :sxfindxreplacex` is legal.

Abbreviations Allowed

- `!!:...` can be shortened to `!:...`
- The `:` can be removed from word designators where it is unambiguous. So `!$` and `!*` are allowed.
- The trailing `/` in a substitution can be omitted if it is unambiguous that the substitution has ended.
- The trailing `?` in a `!?search?` can be omitted for the same reason.
- Any delimiter can be used in a substitution, so `!:sxfindxreplacex` is legal.

Editing Modes

Readline provides editing modes similar to `vi` and `emacs`. Learn one and learn to love it. Most shells and programs have `emacs` as the default.

History Incremental Search

`<C-r>` (vi: `<Esc>/`) brings you to an search of your history. `<C-s>` will reverse the direction of your search (You may need to `stty -ixon`).

Readline Programming in C/C++

C/C++ Readline Library

```
#include <stdio.h>
#include <readline/readline.h>
#include <readline/history.h>

char * readline(const char *prompt);
```

Allocates memory to read a line, reads it from standard input (displaying prompt as the prompt line). Returns the line you read. You really should free the memory it allocated.

Using History Features

```
void using_history(void);
```

Must be called before using history features.

```
int read_history(const char *filename);  
int write_history(const char *filename);
```

For reading/writing saved history. Returns non-zero on failure and sets errno.

```
void add_history(const char *line);
```

Add a line to the history.

```
HIST_ENTRY ** histlst = history_list();  
for (int i = 1; *histlst; i++, histlst++)  
    printf("%d %s\n", i, (*histlst)->line);
```

List history.

Using History Features

```
void using_history(void);
```

Must be called before using history features.

```
int read_history(const char *filename);  
int write_history(const char *filename);
```

For reading/writing saved history. Returns non-zero on failure and sets errno.

```
void add_history(const char *line);
```

Add a line to the history.

```
HIST_ENTRY ** histlst = history_list();  
for (int i = 1; *histlst; i++, histlst++)  
    printf("%d %s\n", i, (*histlst)->line);
```

List history.

Using History Features

```
void using_history(void);
```

Must be called before using history features.

```
int read_history(const char *filename);  
int write_history(const char *filename);
```

For reading/writing saved history. Returns non-zero on failure and sets errno.

```
void add_history(const char *line);
```

Add a line to the history.

```
HIST_ENTRY ** histlst = history_list();  
for (int i = 1; *histlst; i++, histlst++)  
    printf("%d %s\n", i, (*histlst)->line);
```

List history.

Using History Features

```
void using_history(void);
```

Must be called before using history features.

```
int read_history(const char *filename);  
int write_history(const char *filename);
```

For reading/writing saved history. Returns non-zero on failure and sets errno.

```
void add_history(const char *line);
```

Add a line to the history.

```
HIST_ENTRY ** histlst = history_list();  
for (int i = 1; *histlst; i++, histlst++)  
    printf("%d %s\n", i, (*histlst)->line);
```

List history.

History Expansion (for free!)

```
int history_expand(char *string, char **output);
```

Expand string, placing the result into output, a pointer to a string. Returns:

- 0 If no expansions took place
- 1 If expansions did take place
- 1 If there was an error in expansion
- 2 If the line should be displayed, but not executed (:p)

If an error occurred in expansion, then output contains a descriptive error message.

A Complete Example

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5  #include <readline/readline.h>
6  #include <readline/history.h>
7
8  int main(void) {
9      char *line = NULL, *expn = NULL;
10     int status;
11     using_history();
12     for (;;) {
13         free(line), free(expn);
14         line = readline("prompt> ");
15         if (!line) return 0; /* ^D to exit */
16         int expn_result = history_expand(line, &expn);
17         if (expn_result) puts(expn);
18         add_history(expn);
19         if (expn_result == 0 || expn_result == 1) {
20             int pid = fork();
21             if (pid < 0) return 1;
22             if (pid == 0) {
23                 char **arg = history_tokenize(expn);
24                 execvp(*arg, arg);
25                 return 1;
26             }
27             waitpid(pid, &status, 0);
28         }
29     }
30     return 0;
31 }
```

Readline Programming in Python

import readline

To use Readline from Python, type `import readline`, and the `input` function will magically become `readlineified`.

```
import sys
import readline

while True:
    try:
        cmd = input(">>> ")
    except KeyboardInterrupt:
        continue
    except EOFError:
        sys.exit(0)
    print(exec(cmd))
```


Tab Completion

The readline module provides an interface for you to add your own completer:

```
readline.set_completer(function)
```

function should be a function which takes two parameters:

text The current completion text
state 0, 1, ...

Then, set your delimiters and completion keys:

```
readline.set_completer_delims(' ')  
readline.parse_and_bind("tab: complete")
```

Tab Completion

The readline module provides an interface for you to add your own completer:

```
readline.set_completer(function)
```

function should be a function which takes two parameters:

```
text    The current completion text  
state   0, 1, ...
```

Then, set your delimiters and completion keys:

```
readline.set_completer_delims(' ')  
readline.parse_and_bind("tab: complete")
```

Custom Completion in the Wild: iels

```
1 def completer(text, state):
2     def gen():
3         variables = reduce(set.union, map(dict.keys, els.vars), set())
4         for s in '%', '$':
5             for v in variables:
6                 if (s + v).startswith(text):
7                     yield s + v
8         for op in els.operators:
9             if op.startswith(text):
10                 yield op
11         for syntax in 'begin', 'end':
12             if syntax.startswith(text):
13                 yield syntax
14
15     if state == 0:
16         completer.it = gen()
17
18     try:
19         return next(completer.it)
20     except StopIteration:
21         return None
```

Further Resources

More Info

- 1 `man 3 readline`
- 2 `man 3 history`
- 3 `pydoc readline`
- 4 RTFM: Read The *Fine* Manual

Questions?