readline

https://man7.org/linux/man-pages/man3/readline.3.html
https://tiswww.cwru.edu/php/chet/readline/readline.html
https://tiswww.cwru.edu/php/chet/readline/history.html
https://tiswww.cwru.edu/php/chet/readline/rluserman.html

Programming with GNU Readline

Basic Behavior

Many programs provide a command line interface, such as *mail*, *ftp*, and *sh*. For such programs, the default behaviour of Readline is sufficient.

This section describes **how to use** Readline in the simplest way possible, perhaps to replace calls in your code to gets() or fgets().

The function **readline() prints a prompt** and then reads and returns a single line of text from the user.

If **prompt** is **NULL** or **the empty string**, no prompt is displayed.

The line readline returns is allocated with *malloc()*; **the caller should free() the line when it has finished with it**.

The declaration for readline in ANSI C is

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

So, one might say

```
      char *line = readline ("Enter a line: ");
```

in order to read a line of text from the user.

The line returned has the final newline removed, so only the text remains.

If readline encounters an **EOF** while reading the line, **and** the **line is empty** at that point, **then (char *)NULL is returned**. Otherwise, the line is ended just as if a newline had been typed.

Readline **performs** some **expansion** on the prompt before it is displayed on the screen. See the description of **rl_expand_prompt** (*see section 2.4.6 Redisplay*) for additional details, especially if prompt will contain characters that do not consume physical screen space when displayed.

If you want the user to be able to get at the line later, (*with C-p for example*), you must call **add_history()** to save the line away in a history list of such lines.

```
      add_history(line);
```

For full details on the GNU History Library, see the associated manual.

It is preferable to **avoid saving empty lines on the history list**, since users rarely have a burning need to reuse a blank line.

Here is a function which usefully replaces the standard gets() library function, and has the advantage of no static buffer to overflow:

```
/* A static variable for holding the line. */
static char *line_read = (char *)NULL;

/* Read a string, and return a pointer to it.
   Returns NULL on EOF. */
char *rl_gets (void)
{
  /* If the buffer has already been allocated,
     return the memory to the free pool. */
  if (line_read)
    {
      free(line_read);
      line_read = (char *)NULL;
    }

  /* Get a line from the user. */
  line_read = readline("");

  /* If the line has any text in it, save it on the history. */
  if (line_read && *line_read)
    add_history (line_read);

  return (line_read);
}
```

This function gives the user the default behaviour of **TAB completion**: completion on file names. If you do not want Readline to complete on filenames, you can change the binding of the TAB key with **rl_bind_key()**.

```
int rl_bind_key(int key, rl_command_func_t *function);
```

rl_bind_key() takes two arguments: **key** is the character that you want to bind, and **function** is the address of the function to call when key is pressed. Binding TAB to **rl_insert()** makes TAB insert itself.

rl_bind_key() returns non-zero if key is not a valid ASCII character code (between 0 and 255).

Thus, to disable the default TAB behavior, the following suffices:

```
      rl_bind_key('\t', rl_insert);
```

This code should be executed once at the start of your program; you might write a function called **initialize_readline()** which performs this and other desired initializations, such as installing custom completers.

Custom Functions

Readline provides many functions for manipulating the text of the line, but it isn't possible to anticipate the needs of all programs.

This section describes the various functions and variables defined within the Readline library which allow a user program to add customized functionality to Readline.

Before declaring any functions that customize Readline's behavior, or using any functionality Readline provides in other code, an application writer should include the file **<readline/readline.h>** in any file that uses Readline's features. Since some of the definitions in readline.h use the stdio library, the file **<stdio.h>** should be included before readline.h.

readline.h defines a C preprocessor variable that should be treated as an integer, **RL_READLINE_VERSION**, which may be used to conditionally compile application code depending on the installed Readline version. The value is a hexadecimal encoding of the major and minor version numbers of the library, of the form 0xMMmm. MM is the two-digit major version number; mm is the two-digit minor version number. For Readline 4.2, for example, the value of RL_READLINE_VERSION would be 0x0402.

Readline Typedefs

For readability, we declare a number of new object types, all pointers to functions.

The reason for declaring these new types is to make it easier to write code describing pointers to C functions with appropriately prototyped arguments and return values.

For instance, say we want to declare a variable func as a pointer to a function which takes two int arguments and returns an int (this is the type of all of the Readline bindable functions). Instead of the classic C declaration

```
      int (*func)();
```

or the ANSI-C style declaration

```
      int (*func)(int, int);
```

we may write

```
      rl_command_func_t *func;
```

The full list of function pointer types available is

```
      typedef int       rl_command_func_t(int, int);
      typedef char      *rl_compentry_func_t(const char *, int);
      typedef char      **rl_completion_func_t(const char *, int, int);
      typedef char      *rl_quote_func_t(char *, int, char *);
      typedef char      *rl_dequote_func_t(char *, int);
      typedef int       rl_compignore_func_t(char **);
      typedef void      rl_compdisp_func_t(char **, int, int);
      typedef int       rl_hook_func_t(void);
      typedef int       rl_getc_func_t(FILE *);
      typedef int       rl_linebuf_func_t(char *, int);
      typedef int       rl_intfunc_t(int);
      #define rl_ivoidfunc_t rl_hook_func_t
      typedef int       rl_icpfunc_t(char *);
      typedef int       rl_icppfunc_t(char **);
      typedef void      rl_voidfunc_t(void);
      typedef void      rl_vintfunc_t(int);
      typedef void      rl_vcpfunc_t(char *);
      typedef void      rl_vcppfunc_t(char **);
```

Writing a New Function

In order to write new functions for Readline, you need to know the calling conventions for keyboard-invoked functions, and the names of the variables that describe the current state of the line read so far.

The calling sequence for a command foo looks like

```
      int foo (int count, int key)
```

where **count** is the numeric argument (or 1 if defaulted) and **key** is the key that invoked this function.

It is completely up to the function as to what should be done with the numeric argument. Some functions use it as a repeat count, some as a flag, and others to choose alternate behavior (refreshing the current line as opposed to refreshing the screen, for example). Some choose to ignore it. In general, if a function uses the numeric argument as a repeat count, it should be able to do something useful with both negative and positive arguments. At the very least, it should be aware that it can be passed a negative argument.

A command function should return 0 if its action completes successfully, and a value greater than zero if some error occurs. This is the convention obeyed by all of the builtin Readline bindable command functions.

Readline Variables

These variables are available to function writers.

Variable: char * **rl_line_buffer**

This is the line gathered so far. You are welcome to modify the contents of the line. The function **rl_extend_line_buffer** is available to increase the memory allocated to **rl_line_buffer**.

Variable: int **rl_point**

The offset of the current cursor position in **rl_line_buffer(the *point*)**.

Variable: int **rl_end**

The number of characters present in **rl_line_buffer**. When **rl_point** is at the end of the line, **rl_point** and are equal.

Variable: int **rl_mark**

The mark (saved position) in the current line. If set, the mark and point define a *region*.

Variable: int **rl_done**

Setting this to a non-zero value causes Readline to return the current line immediately.

Variable: int **rl_num_chars_to_read**

Setting this to a positive value before calling **readline()** causes Readline to return after accepting that many characters, rather than reading up to a character bound to accept-line.

Variable: int **rl_pending_input**

Setting this to a value makes it the next keystroke read. This is a way to stuff a single character into the input stream.

Variable: int **rl_dispatching**

Set to a non-zero value if a function is being called from a key binding; zero otherwise. Application functions can test this to discover whether they were called directly or by Readline's dispatching mechanism.

Variable: int **rl_erase_empty_line**

Setting this to a non-zero value causes Readline to completely erase the current line, including any prompt, any time a newline is typed as the only character on an otherwise-empty line. The cursor is moved to the beginning of the newly-blank line.

Variable: char * **rl_prompt**

The prompt Readline uses. This is set from the argument to **readline()**, and should not be assigned to directly. The **rl_set_prompt()** function may be used to modify the prompt string after calling **readline()**.

Variable: char * **rl_display_prompt**

The string displayed as the prompt. This is usually identical to **rl_prompt**, but may be changed temporarily by functions that use the prompt string as a message area, such as incremental search.

Variable: int **rl_already_prompted**

If an application wishes to display the prompt itself, rather than have Readline do it the first time **readline()** is called, it should set this variable to a non-zero value after displaying the prompt. The prompt must also be passed as the argument to readline() so the redisplay functions can update the display properly. The calling application is responsible for managing the value; Readline never sets it.

Variable: const char * **rl_library_version**

The version number of this revision of the library.

Variable: int **rl_readline_version**

An integer encoding the current version of the library. The encoding is of the form 0xMMmm, where MM is the two-digit major version number, and mm is the two-digit minor version number. For example, for Readline-4.2, rl_readline_version would have the value 0x0402.

Variable: int **rl_gnu_readline_p**

Always set to 1, denoting that this is GNU readline rather than some emulation.

Variable: const char * **rl_terminal_name**

The terminal type, used for initialization. If not set by the application, Readline sets this to the value of the TERM environment variable the first time it is called.

Variable: const char * **rl_readline_name**

This variable is set to a unique name by each application using Readline. The value allows conditional parsing of the inputrc file.

Variable: FILE * **rl_instream**

The stdio stream from which Readline reads input. If NULL, Readline defaults to stdin.

Variable: FILE * **rl_outstream**

The stdio stream to which Readline performs output. If NULL, Readline defaults to stdout.

Variable: int **rl_prefer_env_winsize**

If non-zero, Readline gives values found in the LINES and COLUMNS environment variables greater precedence than values fetched from the kernel when computing the screen dimensions.

Variable: rl_command_func_t * **rl_last_func**

The address of the last command function Readline executed. May be used to test whether or not a function is being executed twice in succession, for example.

Variable: rl_hook_func_t * **rl_startup_hook**

If non-zero, this is the address of a function to call just before readline prints the first prompt.

Variable: rl_hook_func_t * **rl_pre_input_hook**

If non-zero, this is the address of a function to call after the first prompt has been printed and just before readline starts reading input characters.

Variable: rl_hook_func_t * **rl_event_hook**

If non-zero, this is the address of a function to call periodically when Readline is waiting for terminal input. By default, this will be called at most ten times a second if there is no keyboard input.

Variable: rl_getc_func_t * **rl_getc_function**

If non-zero, Readline will call indirectly through this pointer to get a character from the input stream. By default, it is set to **rl_getc**, the default Readline character input function. In general, an application that sets **rl_getc_function** should consider setting **rl_input_available_hook** as well.

Variable: rl_hook_func_t * **rl_signal_event_hook**

If non-zero, this is the address of a function to call if a read system call is interrupted when Readline is reading terminal input.

Variable: rl_hook_func_t * **rl_input_available_hook**

If non-zero, Readline will use this function's return value when it needs to determine whether or not there is available input on the current input source. The default hook checks **rl_instream**; if an application is using a different input source, it should set the hook appropriately. Readline queries for available input when implementing intra-key-sequence timeouts during input and incremental searches. This may use an application-specific timeout before returning a value; Readline uses the value passed to **rl_set_keyboard_input_timeout()** or the value of the user-settable keyseq-timeout variable. This is designed for use by applications using Readline's callback interface, which may not use the traditional r**ead(2) and file descriptor interface**, or other applications using a different input mechanism. If an application uses an input mechanism or hook that can potentially exceed the value of keyseq-timeout, it should increase the timeout or set this hook appropriately even when not using the callback interface. In general, an application that sets **rl_getc_function** should consider setting **rl_input_available_hook** as well.

Variable: rl_voidfunc_t * **rl_redisplay_function**

If non-zero, Readline will call indirectly through this pointer to update the display with the current contents of the editing buffer. By default, it is set to **rl_redisplay**, the default Readline redisplay function.

Variable: rl_vintfunc_t * **rl_prep_term_function**

If non-zero, Readline will call indirectly through this pointer to initialize the terminal. The function takes a single argument, an int flag that says whether or not to use eight-bit characters. By default, this is set to **rl_prep_terminal**.

Variable: rl_voidfunc_t * **rl_deprep_term_function**

If non-zero, Readline will call indirectly through this pointer to reset the terminal. This function should undo the effects of **rl_prep_term_function**. By default, this is set to **rl_deprep_terminal**.

Variable: Keymap **rl_executing_keymap**

This variable is set to the keymap in which the currently executing readline function was found.

Variable: Keymap **rl_binding_keymap**

This variable is set to the keymap in which the last key binding occurred.

Variable: char * **rl_executing_macro**

This variable is set to the text of any currently-executing macro.

Variable: int **rl_executing_key**

The key that caused the dispatch to the currently-executing Readline function.

Variable: char * **rl_executing_keyseq**

The full key sequence that caused the dispatch to the currently-executing Readline function.

Variable: int **rl_key_sequence_length**

The number of characters in rl_executing_keyseq.

Variable: int **rl_readline_state**

A variable with bit values that encapsulate the current Readline state. A bit is set with the **RL_SETSTATE** macro, and unset with the **RL_UNSETSTATE** macro. Use the **RL_ISSTATE** macro to test whether a particular state bit is set. Current state bits include:

RL_STATE_NONE	Readline has not yet been called, nor has it begun to initialize.
RL_STATE_INITIALIZING	Readline is initializing its internal data structures.
RL_STATE_INITIALIZED	Readline has completed its initialization.
RL_STATE_TERMPREPPED	Readline has modified the terminal modes to do its own input and redisplay.
RL_STATE_READCMD	Readline is reading a command from the keyboard.
RL_STATE_METANEXT	Readline is reading more input after reading the meta-prefix character.
RL_STATE_DISPATCHING	Readline is dispatching to a command.
RL_STATE_MOREINPUT	Readline is reading more input while executing an editing command.
RL_STATE_ISEARCH	Readline is performing an incremental history search.
RL_STATE_NSEARCH	Readline is performing a non-incremental history search.
RL_STATE_SEARCH	Readline is searching backward or forward through the history for a string.
RL_STATE_NUMERICARG	Readline is reading a numeric argument.
RL_STATE_MACROINPUT	Readline is currently getting its input from a previously-defined keyboard macro.
RL_STATE_MACRODEF	Readline is currently reading characters defining a keyboard macro.
RL_STATE_OVERWRITE	Readline is in overwrite mode.
RL_STATE_COMPLETING	Readline is performing word completion.
RL_STATE_SIGHANDLER	Readline is currently executing the readline signal handler.
RL_STATE_UNDOING	Readline is performing an undo.
RL_STATE_INPUTPENDING	Readline has input pending due to a call to rl_execute_next().
RL_STATE_TTYCSAVED	Readline has saved the values of the terminal's special characters.
RL_STATE_CALLBACK	Readline is currently using the alternate (callback) interface.
RL_STATE_VIMOTION	Readline is reading the argument to a vi-mode "motion" command.
RL_STATE_MULTIKEY	Readline is reading a multiple-keystroke command.
RL_STATE_VICMDONCE	Readline has entered vi command (movement) mode at least one time during the current call to readline().
RL_STATE_DONE	Readline has read a key sequence bound to accept-line and is about to return the line to the caller.

Variable: int **rl_explicit_arg**

Set to a non-zero value if an explicit numeric argument was specified by the user. Only valid in a bindable command function.

Variable: int **rl_numeric_arg**

Set to the value of any numeric argument explicitly specified by the user before executing the current Readline function. Only valid in a bindable command function.

Variable: int **rl_editing_mode**

Set to a value denoting Readline's current editing mode. A value of 1 means Readline is currently in emacs mode; 0 means that vi mode is active.

Readline Convenience Functions

Naming a Function

The user can dynamically change the bindings of keys while using Readline. This is done by representing the function with a descriptive name. The user is able to type the descriptive name when referring to the function. Thus, in an init file, one might find

```
Meta-Rubout:
backward-kill-word
```

This binds the keystroke Meta-Rubout to the function *descriptively* named backward-kill-word. You, as the programmer, should bind the functions you write to descriptive names as well. Readline provides a function for doing that:

No function allowed in this section.

Selecting a Keymap

Key bindings take place on a *keymap*. The keymap is the association between the keys that the user types and the functions that get run. You can make your own keymaps, copy existing keymaps, and tell Readline which keymap to use.

No function allowed in this section.

Binding Keys

Key sequences are associated with functions through the keymap. Readline has several internal keymaps: **emacs_standard_keymap**, **emacs_meta_keymap**,

emacs_ctlx_keymap, **vi_movement_keymap**, and **vi_insertion_keymap**. **emacs_standard_keymap** is the default, and the examples in this manual assume that.

Since **readline()** installs a set of default key bindings the first time it is called, there is always the danger that a custom binding installed before the first call to **readline()** will be overridden. An alternate mechanism is to install custom key bindings in an initialization function assigned to the **rl_startup_hook** variable.

No function allowed in this section.

Associating Function Names and Bindings

These functions allow you to find out what keys invoke named functions and the functions invoked by a particular key sequence. You may also associate a new function name with an arbitrary function.

No function allowed in this section.

Allowing Undoing

Supporting the undo command is a painless thing, and makes your functions much more useful. It is certainly easy to try something if you know you can undo it.

If your function simply inserts text once, or deletes text once, and uses **rl_insert_text()** or **rl_delete_text()** to do it, then undoing is already done for you automatically.

If you do multiple insertions or multiple deletions, or any combination of these operations, you should group them together into one operation. This is done with **rl_begin_undo_group()** and **rl_end_undo_group()**.

The types of events that can be undone are:

```
     enum undo_code { UNDO_DELETE, UNDO_INSERT, UNDO_BEGIN, UNDO_END };
```

Notice that **UNDO_DELETE** means to insert some text, and **UNDO_INSERT** means to delete some text. That is, the undo code tells what to undo, not how to undo it. **UNDO_BEGIN** and **UNDO_END** are tags added by **rl_begin_undo_group()** and **rl_end_undo_group()**.

No function allowed in this section.

Redisplay

Function: void **rl_redisplay** *(void)*

Change what's displayed on the screen to reflect the current contents of **rl_line_buffer**.

Function: int **rl_on_new_line** *(void)*

Tell the update functions that we have moved onto a new (empty) line, usually after outputting a newline.

Modifying Text

No function allowed in this section.

Character Input

No function allowed in this section.

Terminal Management

No function allowed in this section.

Utility Functions

Function: void **rl_replace_line** *(const char *text, int clear_undo)*

Replace the contents of **rl_line_buffer** with **text**. The **point** and **mark** are preserved, if possible. If **clear_undo** is non-zero, the undo list associated with the current line is cleared.

Miscellaneous Functions

Function: void **rl_clear_history** *(void)*

Clear the history list by deleting all of the entries, in the same manner as the History library's **clear_history()** function. This differs from **clear_history** because it frees private data Readline saves in the history list.

Alternate Interface

An alternate interface is available to plain **readline()**. Some applications need to interleave keyboard I/O with file, device, or window system I/O, typically by using a main loop to select() on various file descriptors. To accommodate this need, readline can also be invoked as a `callback' function from an event loop. There are functions available to make this easy.

No function allowed in this section.

A Readline Example

Here is a function which changes lowercase characters to their uppercase equivalents, and uppercase characters to lowercase. If this function was bound to `M-c', then typing `M-c' would change the case of the character under point. Typing `M-1 0 M-c' would change the case of the following 10 characters, leaving the cursor on the last character changed.

```
/* Invert the case of the COUNT following characters. */
int invert_case_line (int count, int key) {
  register int start, end, i;

  start = rl_point;
  if (rl_point >= rl_end)
    return (0);
  if (count < 0)
    {
      direction = -1;
      count = -count;
    }
  else
    direction = 1;
  /* Find the end of the range to modify. */
  end = start + (count * direction);
  /* Force it to be within range. */
  if (end > rl_end)
    end = rl_end;
  else if (end < 0)
    end = 0;
  if (start == end)
    return (0);
  if (start > end)
    {
      int temp = start;
      start = end;
      end = temp;
    }
  /* Tell readline that we are modifying the line,
     so it will save the undo information. */
  rl_modifying(start, end);
  for (i = start; i != end; i++)
    {
      if (_rl_uppercase_p(rl_line_buffer[i]))
        rl_line_buffer[i] = _rl_to_lower(rl_line_buffer[i]);
      else if (_rl_lowercase_p(rl_line_buffer[i]))
        rl_line_buffer[i] = _rl_to_upper(rl_line_buffer[i]);
    }
  /* Move point to on top of the last character changed. */
  rl_point = (direction == 1) ? end - 1 : start;
  return (0);
}
```

Alternate Interface Example

Here is a complete program that illustrates Readline's alternate interface. It reads lines from the terminal and displays them, providing the standard history and TAB completion functions. It understands the EOF character or "exit" to exit the program.

```
  /* Standard include files. stdio.h is required. */
  #include <stdlib.h>
  #include <string.h>
  #include <unistd.h>
  #include <locale.h>
  /* Used for select(2) */
  #include <sys/types.h>
  #include <sys/select.h>
  #include <signal.h>
  #include <stdio.h>
  /* Standard readline include files. */
  #include <readline/readline.h>
  #include <readline/history.h>

      static void        cb_linehandler(char *);
      static void        sighandler(int);
      int                running;
      int                sigwinch_received;
      const char         *prompt = "rltest$ ";
      /* Handle SIGWINCH and window size changes when readline is not
active and reading a character. */
      static void        sighandler(int sig)
      {
            sigwinch_received = 1;
      }
      /* Callback function called for each line when accept-line executed,
EOF seen, or EOF character read.  This sets a flag and returns; it could
also call exit(3). */
      static void       cb_linehandler (char *line)
      {
            /* Can use ^D (stty eof) or `exit' to exit. */
            if (line == NULL || strcmp(line, "exit") == 0)
            {
                  if (line == 0)
                        printf("\n");
                  printf("exit\n");
                  /* This function needs to be called to reset the
terminal settings, and calling it from the line handler keeps one extra
prompt from being displayed. */
                  rl_callback_handler_remove();
                  running = 0;
            }
            else
            {
                  if (*line)
                        add_history(line);
                  printf("input line: %s\n", line);
                  free(line);
            }
      }

      int               main(int c, char **v)
      {
            fd_set fds;
            int r;

            /* Set the default locale values according to environment
variables. */
            setlocale(LC_ALL, "");
            /* Handle window size changes when readline is not active and
reading characters. */
            signal(SIGWINCH, sighandler);
            /* Install the line handler. */
            rl_callback_handler_install(prompt, cb_linehandler);
            /* Enter a simple event loop.  This waits until something is
available to read on readline's input stream (defaults to standard input)
and calls the builtin character read callback to read it. It does not have
to modify the user's terminal settings. */
            running = 1;
            while (running)
            {
                  FD_ZERO(&fds);
                  FD_SET(fileno(rl_instream), &fds);
                  r = select(FD_SETSIZE, &fds, NULL, NULL, NULL);
                  if (r < 0 && errno != EINTR)
                  {
                        perror("rltest: select");
                        rl_callback_handler_remove();
                        break;
                  }
                  if (sigwinch_received)
                  {
                        rl_resize_terminal();
                        sigwinch_received = 0;
                  }
                  if (r < 0)
                        continue;
                  if (FD_ISSET(fileno(rl_instream), &fds))
                        rl_callback_read_char();
            }
            printf("rltest: Event loop has exited\n");
            return 0;
      }
```

Readline Signal Handling

Signals are asynchronous events sent to a process by the Unix kernel, sometimes on behalf of another process. They are intended to indicate exceptional events, like a user pressing the interrupt key on his terminal, or a network connection being broken. There is a class of signals that can be sent to the process currently reading input from the keyboard. Since Readline changes the terminal attributes when it is called, it needs to perform special processing when such a signal is received in order to restore the terminal to a sane state, or provide application writers with functions to do so manually.

Readline contains an internal signal handler that is installed for a number of signals (**SIGINT**, **SIGQUIT**, **SIGTERM**, **SIGHUP**, **SIGALRM**, **SIGTSTP**, **SIGTTIN**, and **SIGTTOU**). When one of these signals is received, the signal handler will reset the terminal attributes to those that were in effect before **readline()** was called, reset the signal handling to what it was before readline() was called, and resend the signal to the calling application. If and when the calling application's signal handler returns, Readline will reinitialize the terminal and continue to accept input.

When a **SIGINT** is received, the Readline signal handler performs some additional work, which will cause any partially-entered line to be aborted.

There is an additional Readline signal handler, for **SIGWINCH**, which the kernel sends to a process whenever the terminal's size changes (for example, if a user resizes an xterm). The Readline **SIGWINCH** handler updates Readline's internal screen size information, and then calls any **SIGWINCH** signal handler the calling application has installed. Readline calls the application's **SIGWINCH** signal handler without resetting the terminal to its original state. If the application's signal handler does more than update its idea of the terminal size and return (for example, a longjmp back to a main processing loop), it *must* call **rl_cleanup_after_signal()**, to restore the terminal state.

When an application is using the callback interface, Readline installs signal handlers only for the duration of the call to **rl_callback_read_char**. Applications using the callback interface should be prepared to clean up Readline's state if they wish to handle the signal before the line handler completes and restores the terminal state.

If an application using the callback interface wishes to have Readline install its signal handlers at the time the application calls **rl_callback_handler_install** and remove them only when a complete line of input has been read, it should set the **rl_persistent_signal_handlers** variable to a non-zero value. This allows an application to defer all of the handling of the signals Readline catches to Readline. Applications should use this variable with care; it can result in Readline catching signals and not acting on them (or allowing the application to react to them) until the application calls **rl_callback_read_char**. This can result in an application becoming less responsive to keyboard signals like **SIGINT**. If an application does not want or need to perform any signal handling, or does not need to do any processing between calls to r**l_callback_read_char**, setting this variable may be desirable.

Readline provides two variables that allow application writers to control whether or not it will catch certain signals and act on them when they are received. It is important that applications change the values of these variables only when calling **readline()**, not in a signal handler, so Readline's internal signal state is not corrupted.

No function allowed in this section.

Custom Completers

Typically, a program that reads commands from the user has a way of disambiguating commands and data. If your program is one of these, then it can provide completion for commands, data, or both. The following sections describe how your program and Readline cooperate to provide this service.

How Completing Works

In order to complete some text, the full list of possible completions must be available. That is, it is not possible to accurately expand a partial word without knowing all of the possible words which make sense in that context. The Readline library provides the user interface to completion, and two of the most common completion functions: **filename** and **username**. For completing other types of text, you must write your own completion function. This section describes exactly what such functions must do, and provides an example.

There are three major functions used to perform completion:

1. The user-interface function **rl_complete()**. This function is called with the same arguments as other bindable Readline functions: **count** and **invoking_key**. It isolates the word to be completed and calls **rl_completion_matches()** to generate a list of possible completions. It then either lists the possible completions, inserts the possible completions, or actually performs the completion, depending on which behavior is desired.
2. The internal function **rl_completion_matches()** uses an application-supplied *generator* function to generate the list of possible matches, and then returns the array of these matches. The caller should place the address of its generator function in **rl_completion_entry_function**.
3. The generator function is called repeatedly from **rl_completion_matches()**, returning a string each time. The arguments to the generator function are text and state. text is the partial word to be completed. state is zero the first time the function is called, allowing the generator to perform any necessary initialization, and a positive non-zero integer for each subsequent call. The generator function returns **(char *)NULL** to inform r**l_completion_matches()** that there are no more possibilities left. Usually the generator function computes the list of possible completions when state is zero, and returns them one at a time on subsequent calls. Each string the generator function returns as a match must be allocated with *malloc()*; Readline frees the strings when it has finished with them. Such a generator function is referred to as an *application-specific completion function*.

Completion Functions

No function allowed in this section.

Completion Variables

No function allowed in this section.

A Short Completion Example

Here is a small application demonstrating the use of the GNU Readline library. It is called **fileman**, and the source code resides in `**examples/fileman.c**'. This sample application provides completion of command names, line editing features, and access to the history list.

```
/* fileman.c -- A tiny application which demonstrates how to use the
   GNU Readline library.  This application interactively allows users
   to manipulate files and their modes. */

#ifdef HAVE_CONFIG_H
#  include <config.h>
#endif
#include <sys/types.h>
#ifdef HAVE_SYS_FILE_H
#  include <sys/file.h>
#endif
#include <sys/stat.h>
#ifdef HAVE_UNISTD_H
#  include <unistd.h>
#endif
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#if defined (HAVE_STRING_H)
#  include <string.h>
#else /* !HAVE_STRING_H */
#  include <strings.h>
#endif /* !HAVE_STRING_H */
#ifdef HAVE_STDLIB_H
#  include <stdlib.h>
#endif
#include <time.h>
#include <readline/readline.h>
#include <readline/history.h>
```

```
      extern char *xmalloc PARAMS((size_t));

      /* The names of functions that actually do the manipulation. */
      int com_list PARAMS((char *));
      int com_view PARAMS((char *));
      int com_rename PARAMS((char *));
      int com_stat PARAMS((char *));
      int com_pwd PARAMS((char *));
      int com_delete PARAMS((char *));
      int com_help PARAMS((char *));
      int com_cd PARAMS((char *));
      int com_quit PARAMS((char *));

      /* A structure which contains information on the commands this
program can understand. */
      typedef struct {
            char         *name; /* User printable name of the function. */
            rl_icpfunc_t *func; /* Function to call to do the job. */
            char         *doc;  /* Documentation for this function.  */
      } COMMAND;

      COMMAND commands[] = {
            { "cd", com_cd, "Change to directory DIR" },
            { "delete", com_delete, "Delete FILE" },
            { "help", com_help, "Display this text" },
            { "?", com_help, "Synonym for `help'" },
            { "list", com_list, "List files in DIR" },
            { "ls", com_list, "Synonym for `list'" },
            { "pwd", com_pwd, "Print the current working directory" },
            { "quit", com_quit, "Quit using Fileman" },
            { "rename", com_rename, "Rename FILE to NEWNAME" },
            { "stat", com_stat, "Print out statistics on FILE" },
            { "view", com_view, "View the contents of FILE" },
            { (char *)NULL, (rl_icpfunc_t *)NULL, (char *)NULL }
      };

      /* Forward declarations. */
      char        *stripwhite();
      COMMAND     *find_command();
```

```
	/* The name of this program, as taken from argv[0]. */
	char	*progname;

	/* When non-zero, this global means the user is done using this
program. */
	int done;

	char	*dupstr(char *s)
	{
		char *r;

		r = xmalloc(strlen(s) + 1);
		strcpy(r, s);
		return (r);
	}

	int	main(int argc, char **argv)
	{
		char *line, *s;

		progname = argv[0];
		initialize_readline();  /* Bind our completer. */
		/* Loop reading and executing lines until the user quits. */
		for ( ; done == 0; )
		{
			line = readline ("FileMan: ");
			if (!line)
				break;
			/* Remove leading and trailing whitespace from the line.
Then, if there is anything left, add it to the history list and execute
it. */
			s = stripwhite(line);
			if (*s)
			{
				add_history(s);
				execute_line(s);
			}
			free(line);
		}
		exit(0);
	}

	/* Execute a command line. */
	int execute_line(char *line)
	{
		register int i;
		COMMAND *command;
		char *word;

		/* Isolate the command word. */
		i = 0;
		while (line[i] && whitespace(line[i]))
			i++;
		word = line + i;
		while (line[i] && !whitespace(line[i]))
			i++;
		if (line[i])
			line[i++] = '\0';
		command = find_command(word);
		if (!command)
		{
			fprintf(stderr, "%s: No such command for FileMan.\n", 
word);
			return (-1);
		}
		/* Get argument to command, if any. */
		while (whitespace(line[i]))
			i++;
		word = line + i;
		/* Call the function. */
		return ((*(command->func))(word));
	}

	/* Look up NAME as the name of a command, and return a pointer to 
that command.  Return a NULL pointer if NAME isn't a command name. */
	COMMAND *find_command(char *name)
	{
```

```
            register int i;

            for (i = 0; commands[i].name; i++)
                  if (strcmp(name, commands[i].name) == 0)
                        return (&commands[i]);
            return ((COMMAND *)NULL);
      }

      /* Strip whitespace from the start and end of STRING.  Return a
pointer into STRING. */
      char *stripwhite(char *string)
      {
            register char *s, *t;

            for (s = string; whitespace(*s); s++)
                  ;
            if (*s == 0)
                  return (s);
            t = s + strlen(s) - 1;
            while (t > s && whitespace(*t))
                  t--;
            *++t = '\0';
            return s;
      }

  /* ****************************************************************** */
  /*                                                                    */
  /*                  Interface to Readline Completion                  */
  /*                                                                    */
  /* ****************************************************************** */

      char *command_generator PARAMS((const char *, int));
      char **fileman_completion PARAMS((const char *, int, int));

      /* Tell the GNU Readline library how to complete. We want to try to
complete on command names if this is the first word in the line, or on
filenames if not. */
      void  initialize_readline(void)
      {
            /* Allow conditional parsing of the ~/.inputrc file. */
            rl_readline_name = "FileMan";
            /* Tell the completer that we want a crack first. */
            rl_attempted_completion_function = fileman_completion;
      }

      /* Attempt to complete on the contents of TEXT. START and END bound
the region of rl_line_buffer that contains the word to complete. TEXT is
the word to complete. We can use the entire contents of rl_line_buffer in
case we want to do some simple parsing. Return the array of matches, or
NULL if there aren't any. */
      char  **fileman_completion (const char *text, int start,int end)
      {
            char **matches;

            matches = (char **)NULL;
            /* If this word is at the start of the line, then it is a
command to complete.  Otherwise it is the name of a file in the current
directory. */
            if (start == 0)
                  matches = rl_completion_matches(text,
command_generator);
            return (matches);
      }

      /* Generator function for command completion.  STATE lets us know
whether to start from scratch; without any state (i.e. STATE == 0), then
we start at the top of the list. */
      char  *command_generator (const char *text, int state)
      {
            static int list_index, len;
            char *name;

            /* If this is a new word to complete, initialize now. This
includes saving the length of TEXT for efficiency, and initializing the
index variable to 0. */
            if (!state)
            {
                  list_index = 0;
                  len = strlen(text);
            }
            /* Return the next name which partially matches from the
command list. */
            while (name = commands[list_index].name)
            {
                  list_index++;

                  if (strncmp(name, text, len) == 0)
                        return (dupstr(name));
          }
            /* If no names matched, then return NULL. */
            return ((char *)NULL);
      }

  /* **************************************************************** */
  /*                                                                  */
  /*                      FileMan Commands                            */
  /*                                                                  */
  /* **************************************************************** */

      /* String to pass to system(). This is for the LIST, VIEW and RENAME
commands. */
      static char syscom[1024];

      /* List the file(s) named in arg. */
      int   com_list(char *arg)
      {
            if (!arg)
                  arg = "";
            sprintf(syscom, "ls -FClg %s", arg);
            return (system(syscom));
      }

      int   com_view(char *arg)
      {
            if (!valid_argument("view", arg))
                  return 1;
         #if defined (__MSDOS__)
            /* more.com doesn't grok slashes in pathnames */
            sprintf(syscom, "less %s", arg);
         #else
            sprintf(syscom, "more %s", arg);
         #endif
            return (system (syscom));
      }

      int   com_rename(char *arg)
      {
            too_dangerous("rename");
            return (1);
      }

      int com_stat(char *arg)
      {
            struct stat finfo;

            if (!valid_argument("stat", arg))
                  return (1);
            if (stat(arg, &finfo) == -1)
            {
                  perror(arg);
                  return (1);
            }
            printf("Statistics for `%s':\n", arg);
            printf("%s has %d link%s, and is %d byte%s in length.\n",
arg, finfo.st_nlink, (finfo.st_nlink == 1) ? "" : "s", finfo.st_size,
(finfo.st_size == 1) ? "" : "s");
            printf("Inode Last Change at: %s", ctime (&finfo.st_ctime));
            printf("      Last access at: %s", ctime (&finfo.st_atime));
            printf("    Last modified at: %s", ctime (&finfo.st_mtime));
            return (0);
      }

      int   com_delete(char *arg)
      {
            too_dangerous("delete");
            return (1);
      }
```

```
	/* Print out help for ARG, or for all of the commands if ARG is not 
present. */
	int	com_help(char *arg)
	{
		register int i;
		int printed = 0;

		for (i = 0; commands[i].name; i++)
		{
			if (!*arg || (strcmp(arg, commands[i].name) == 0))
			{
				printf("%s\t\t%s.\n", commands[i].name, 
commands[i].doc);
				printed++;
			}
		}
		if (!printed)
		{
			printf("No commands match `%s'.  Possibilities are:\n", 
arg);
			for (i = 0; commands[i].name; i++)
			{
				/* Print in six columns. */
				if (printed == 6)
				{
					printed = 0;
					printf("\n");
				}
				printf("%s\t", commands[i].name);
				printed++;
			}
			if (printed)
				printf("\n");
		}
		return (0);
	}

	/* Change to the directory ARG. */
	void	com_cd(char *arg)
```

```c
      {
            if (chdir(arg) == -1)
            {
                  perror(arg);
                  return 1;
            }
            com_pwd("");
            return (0);
      }

      /* Print out the current working directory. */
      void  com_pwd(char *ignore)
      {
            char dir[1024], *s;

            s = getcwd(dir, sizeof(dir) - 1);
            if (s == 0)
            {
                  printf("Error getting pwd: %s\n", dir);
                  return 1;
            }
            printf("Current directory is %s\n", dir);
            return 0;
      }

      /* The user wishes to quit using this program. Just set DONE
non-zero. */
      void  com_quit(char *arg)
      {
            done = 1;
            return (0);
      }

      /* Function which tells you that you can't do this. */
      void  too_dangerous(char *caller)
      {
            fprintf(stderr, "%s: Too dangerous for me to distribute.
Write it yourself.\n", caller);
      }
      /* Return non-zero if ARG is a valid argument for CALLER, else print
an error message and return zero. */
      int   valid_argument (char *caller, char *arg)
      {
            if (!arg || !*arg)
            {
                  fprintf(stderr, "%s: Argument required.\n", caller);
                  return (0);
            }
            return (1);
      }
```

GNU History Library

This document describes the GNU History library, a programming tool that provides a consistent user interface for recalling lines of previously typed input.

Using History Interactively

This chapter describes how to use the GNU History Library interactively, from a user's standpoint. It should be considered a user's guide. For information on using the GNU History Library in your own programs.

History Expansion

The History library provides a history expansion feature that is similar to the history expansion provided by csh. This section describes the syntax used to manipulate the history information.

History expansions introduce words from the history list into the input stream, making it easy to repeat commands, insert the arguments to a previous command into the current input line, or fix errors in previous commands quickly.

History expansion takes place in two parts. The first is to determine which line from the history list should be used during substitution. The second is to select portions of that line for inclusion into the current one. The line selected from the history is called the *event*, and the portions of that line that are acted upon are called *words*. Various *modifiers* are available to manipulate the selected words. The line is broken into words in the same fashion that Bash does, so that several words surrounded by quotes are considered one word. History expansions are introduced by the appearance of the history expansion character, which is `!' by default.

History expansion implements shell-like quoting conventions: a backslash can be used to remove the special handling for the next character; single quotes enclose verbatim sequences of characters, and can be used to inhibit history expansion; and characters enclosed within double quotes may be subject to history expansion, since backslash can escape the history expansion character, but single quotes may not, since they are not treated specially within double quotes.

Event Designators

An event designator is a reference to a command line entry in the history list. Unless the reference is absolute, events are relative to the current position in the history list.

!	Start a history substitution, except when followed by a space, tab, the end ofthe line, or `='.
!n	Refer to command line n.
!-n	Refer to the command n lines back.
!!	Refer to the previous command. This is a synonym for `!-1'.
!string	Refer to the most recent command preceding the current position in the history list starting with string.
!?string[?]	Refer to the most recent command preceding the current position in the history list containing string. The trailing `?' may be omitted if the string is followed immediately by a newline. If string is missing, the string from the most recent search is used; it is an error if there is no previous search string.
^string1^string2^	Quick Substitution. Repeat the last command, replacing string1 with string2. Equivalent to !!:s^string1^string2^.
!#	The entire command line typed so far.

Word Designators

Word designators are used to select desired words from the event. A `:' separates the event specification from the word designator. It may be omitted if the word designator begins with a `^', `$', `*', `-', or `%'. Words are numbered from the beginning of the line, with the first word being denoted by 0 (zero). Words are inserted into the current line separated by single spaces.

For example,

!!	designates the preceding command. When you type this, the preceding command is repeated in toto.

!!:$	designates the last argument of the preceding command. This may be shortened to !$.
!fi:2	designates the second argument of the most recent command starting with the letters fi.

Here are the word designators:

0 (zero)	The 0th word. For many applications, this is the command word.
n	The nth word.
^	The first argument; that is, word 1.
$	The last argument.
%	The first word matched by the most recent \`?string?' search, if the search string begins with a character that is part of a word.
x-y	A range of words; \`-y' abbreviates \`0-y'.
*****	All of the words, except the 0th. This is a synonym for \`1-$'. It is not an error to use \`*' if there is just one word in the event; the empty string is returned in that case.
x*	Abbreviates \`x-$'
x-	Abbreviates \`x-$' like \`x*', but omits the last word. If \`x' is missing, it defaults to 0.

If a word designator is supplied without an event specification, the previous command is used as the event.

Modifiers

After the optional word designator, you can add a sequence of one or more of the following modifiers, each preceded by a \`:'. These modify, or edit, the word or words selected from the history event.

h	Remove a trailing pathname component, leaving only the head.
t	Remove all leading pathname components, leaving the tail.
r	Remove a trailing suffix of the form \`.suffix', leaving the basename.
e	Remove all but the trailing suffix.

p	Print the new command but do not execute it.
s/old/new/	Substitute new for the first occurrence of old in the event line. Any character may be used as the delimiter in place of `/'. The delimiter may be quoted in old and new with a single backslash. If `&' appears in new, it is replaced by old. A single backslash will quote the `&'. If old is null, it is set to the last old substituted, or, if no previous history substitutions took place, the last string in a !?string[?] search. If new is is null, each matching old is deleted. The final delimiter is optional if it is the last character on the input line.
&	Repeat the previous substitution.
g, a	Cause changes to be applied over the entire event line. Used in conjunction with `s', as in gs/old/new/, or with `&'.
G	Apply the following `s' or `&' modifier once to each word in the event.

Programming with GNU History

This chapter describes how to interface programs that you write with the GNU History Library. It should be considered a technical guide. For information on the interactive use of GNU History.

Introduction to History

Many programs read input from the user a line at a time. The GNU History library is able to keep track of those lines, associate arbitrary data with each line, and utilize information from previous lines in composing new ones.

A programmer using the History library has available functions for remembering lines on a history list, associating arbitrary data with a line, removing lines from the list, searching through the list for a line containing an arbitrary text string, and referencing any line in the list directly. In addition, a history *expansion* function is available which provides for a consistent user interface across different programs.

The user using programs written with the History library has the benefit of a consistent user interface with a set of well-known commands for manipulating the text of previous lines and using that text in new commands. The basic history manipulation commands are similar to the history substitution provided by csh.

The programmer can also use the Readline library, which includes some history manipulation by default, and has the added advantage of command line editing.

Before declaring any functions using any functionality the History library provides in other code, an application writer should include the file <readline/history.h> in any file that uses the History library's features. It supplies extern declarations for all of the library's public functions and variables, and declares all of the public data structures.

History Storage

The history list is an array of history entries. A history entry is declared as follows:

```
      typedef void *histdata_t;

      typedef struct _hist_entry {
        char *line;
        char *timestamp;
        histdata_t data;
      } HIST_ENTRY;
```

The history list itself might therefore be declared as

```
      HIST_ENTRY **the_history_list;
```

The state of the History library is encapsulated into a single structure:

```
  /*
   * A structure used to pass around the current state of the history.
   */
      typedef struct _hist_state {
        HIST_ENTRY **entries; /* Pointer to the entries themselves. */
        int offset;         /* The location pointer within this array. */
        int length;           /* Number of elements within this array. */
        int size;           /* Number of slots allocated to this array. */
        int flags;
      } HISTORY_STATE;
```

If the flags member includes **HS_STIFLED**, the history has been stifled.

History Functions

This section describes the calling sequence for the various functions exported by the GNU History library.

Initializing History and State Management

This section describes functions used to initialize and manage the state of the History library when you want to use the history functions in your program.

No function allowed in this section.

History List Management

These functions manage individual entries on the history list, or set parameters managing the list itself.

Function: void **add_history** *(const char *string)*

Place string at the end of the history list. The associated data field (if any) is set to NULL. If the maximum number of history entries has been set using stifle_history(), and the new number of history entries would exceed that maximum, the oldest history entry is removed.

Information About the History List

These functions return information about the entire history list or individual list entries.

No function allowed in this section.

Moving Around the History List

These functions allow the current index into the history list to be set or changed.

No function allowed in this section.

Searching the History List

These functions allow searching of the history list for entries containing a specific string. Searching may be performed both forward and backward from the current history position. The search may be *anchored*, meaning that the string must match at the beginning of the history entry.

No function allowed in this section.

Managing the History File

The History library can read the history from and write it to a file. This section documents the functions for managing a history file.

No function allowed in this section.

History Expansion

These functions implement history expansion.

No function allowed in this section.

History Variables

This section describes the externally-visible variables exported by the GNU History Library.

Variable: int **history_base**

The logical offset of the first entry in the history list.

Variable: int **history_length**

The number of entries currently stored in the history list.

Variable: int **history_max_entries**

The maximum number of history entries. This must be changed using stifle_history().

Variable: int **history_write_timestamps**

If non-zero, timestamps are written to the history file, so they can be preserved between sessions. The default value is 0, meaning that timestamps are not saved.

The current timestamp format uses the value of history_comment_char to delimit timestamp entries in the history file. If that variable does not have a value (the default), timestamps will not be written.

Variable: char **history_expansion_char**

The character that introduces a history event. The default is `!'. Setting this to 0 inhibits history expansion.

Variable: char **history_subst_char**

The character that invokes word substitution if found at the start of a line. The default is `^'.

Variable: char **history_comment_char**

During tokenization, if this character is seen as the first character of a word, then it and all subsequent characters up to a newline are ignored, suppressing history expansion for the remainder of the line. This is disabled by default.

Variable: char * **history_word_delimiters**

The characters that separate tokens for history_tokenize(). The default value is " \t\n()<>;&|".

Variable: char * **history_search_delimiter_chars**

The list of additional characters which can delimit a history search string, in addition to space, TAB, `:' and `?' in the case of a substring search. The default is empty.

Variable: char * **history_no_expand_chars**

The list of characters which inhibit history expansion if found immediately following history_expansion_char. The default is space, tab, newline, carriage return, and `='.

Variable: int **history_quotes_inhibit_expansion**

If non-zero, the history expansion code implements shell-like quoting: single-quoted words are not scanned for the history expansion character or the history comment character, and double-quoted words may have history expansion performed, since single quotes are not special within double quotes. The default value is 0.

Variable: int **history_quoting_state**

An application may set this variable to indicate that the current line being expanded is subject to existing quoting. If set to `'', the history expansion function will assume that the line is single-quoted and inhibit expansion until it reads an unquoted closing single quote; if set to `"', history expansion will assume the line is double quoted until it reads an unquoted closing double quote. If set to zero, the default, the history expansion function will assume the line is not quoted and treat quote characters within the line as described above. This is only effective if history_quotes_inhibit_expansion is set.

Variable: rl_linebuf_func_t * **history_inhibit_expansion_function**

This should be set to the address of a function that takes two arguments: a char * (string) and an int index into that string (i). It should return a non-zero value if the history expansion starting at string[i] should not be performed; zero if the expansion should be done. It is intended for use by applications like Bash that use the history expansion character for additional purposes. By default, this variable is set to NULL.

History Programming Example

The following program demonstrates simple use of the GNU History Library.

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

```
int     main (int argc, char **argv)
{
        char line[1024], *t;
        int len, done = 0;

        line[0] = 0;
        using_history();
        while (!done)
        {
                printf("history$ ");
                fflush(stdout);
                t = fgets(line, sizeof(line) - 1, stdin);
                if (t && *t)
                {
                        len = strlen(t);
                        if (t[len - 1] == '\n')
                                t[len - 1] = '\0';
                }
                if (!t)
                        strcpy(line, "quit");
                if (line[0])
                {
                        char *expansion;
                        int result;

                        result = history_expand(line, &expansion);
                        if (result)
                                fprintf(stderr, "%s\n", expansion);
                        if (result < 0 || result == 2)
                        {
                                free(expansion);
                                continue;
                        }
                        add_history(expansion);
                        strncpy(line, expansion, sizeof(line) - 1);
                        free(expansion);
                }
                if (strcmp(line, "quit") == 0)
                done = 1;
            else if (strcmp(line, "save") == 0)
                write_history("history_file");
            else if (strcmp(line, "read") == 0)
                read_history("history_file");
            else if (strcmp(line, "list") == 0)
            {
                register HIST_ENTRY **the_list;
                register int i;

                the_list = history_list();
                if (the_list)
                    for (i = 0; the_list[i]; i++)
                        printf("%d: %s\n", i + history_base,
the_list[i]->line);
            }
            else if (strncmp(line, "delete", 6) == 0)
            {
                int which;

                if ((sscanf(line + 6, "%d", &which)) == 1)
                {
                    HIST_ENTRY *entry = remove_history(which);
                    if (!entry)
                        fprintf(stderr, "No such entry %d\n",
which);
                    else
                    {
                        free(entry->line);
                        free(entry);
                    }
                }
                else
                {
                    fprintf(stderr, "non-numeric arg given to
`delete'\n");
                }
            }
        }
```

```
    }
```

https://docs.google.com/document/d/1lfbserdSA1FjrMhEdBWeH75ZLgETlERDnh60j7pp1KU/edit