✓Minishell Functions

https://www.notion.so/Minishell-Materials-7bbd45a806e04395ab578ca3f805806c
https://harm-smits.github.io/42docs/projects/minishell
https://www.cs.purdue.edu/homes/grr/SystemsProgrammingBook/Book/Chapter5-WritingYourOwnShell.pdf
https://brennan.io/2015/01/16/write-a-shell-in-c/
https://www.youtube.com/playlist?list=PLfqABt5AS4FkW5mOn2Tn9ZZLLDwA3kZUY
https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html
https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html
http://www.cems.uwe.ac.uk/~irjohnso/coursenotes/lrc/system/shell/cs3.htm
https://www.howtogeek.com/719058/how-to-use-here-documents-in-bash-on-linux/
https://awesomeopensource.com/project/kukinpower/minishell?category
https://www.cyberciti.biz/faq/linux-bash-exit-status-set-exit-statusin-bash/
https://www.youtube.com/watch?v=tcYo6hipaSA
https://www.geeksforgeeks.org/exit-status-child-process-linux/
https://www.cyberciti.biz/faq/linux-bash-exit-status-set-exit-statusin-bash/

https://github.com/Swoorup/mysh
https://github.com/ska42/minishell
https://github.com/mittsh/42sh

https://github.com/potatokuka/mini_tester

readline **#include <stdio.h>** **#include <readline/readline.h>** **#include <readline/history.h>** ***char** **readline (const char *prompt);***
The function readline() prints a prompt *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. For example —> 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.
https://tiswww.case.edu/php/chet/readline/readline.html

rl_clear_history

void rl_clear_history *(void);*
Enable an *active* mark. When this is enabled, the text between point and mark

(the *region*) is displayed in the terminal's standout mode (a *face*). This is called by various readline functions that set the mark and insert text, and is available for applications to call.

rl_on_new_line

int rl_on_new_line *(void);*

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

rl_replace_line

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.

rl_redisplay

void **rl_redisplay** *(void)***;**

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

add_history

void(?) add_history (line);

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.

printf **#include <stdio.h>**
int printf(cont char *str, ...);

The function printf produces specific output according to a format specifier.

malloc **#include <stdlib.h>**
void *malloc(size_t size);

allocate memory with a given size on the heap.

free **#include <stdlib.h>**
void free(void *ptr);

free previously allocated memory.

write **#include <unistd.h>**
ssize_t write(int fildes, const void *buf, size_t nbyte);
write() attempts to write nbyte of data to the object referenced by the descriptor fildes from the buffer pointed to by buf

access **#include <unistd.h>**
int access(const char *path, int mode);
The access() system call checks the accessibility of the file named by the path argument for the access permissions indicated by the mode argument. The value of mode is either the bitwise-inclusive OR of the access permissions to be checked (R_OK for read permission, W_OK for write permission, and X_OK for execute/search permission), or the existence test (F_OK).

open **#include <fcntl.h>**
int open(const char *path, int oflag, ...);
Path : path to file which you want to use
use absolute path begin with “/”, when you are not work in same directory of file.
Use relative path which is only file name with extension, when you are work in same directory of file.
flags : How you like to use
O_RDONLY: read only, O_WRONLY: write only, O_RDWR: read and write, O_CREATE: create file if it doesn’t exist, O_EXCL: prevent creation if it already exists

read **#include <unistd.h>**
ssize_t read(int fildes, void *buf, size_t nbyte);
read from a file-descriptor to a string with a given buffersize.

close **#include <unistd.h>**
int close(int fildes);
Tells the operating system you are done with a file descriptor and Close the file which pointed by fd.

fork **#include <unistd.h>**
pid_t fork(void);
Fork system call is used for creating a new process, which is called *child process*, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call. A child process

uses the same pc(program counter), same CPU registers, same open files which use in the parent process.
It takes no parameters and returns an integer value. Below are different values returned by fork().

wait **#include <sys/wait.h>**
pid_t wait(int *stat_loc);

A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent *continues* its execution after wait system call instruction.
Child process may terminate due to any of these:

- It calls exit();
- It returns (an int) from main
- It receives a signal (from the OS or another process) whose default action is to terminate.

waitpid **#include <sys/wait.h>**
pid_t waitpid(pid_t pid, int *stat_loc, int options);

waitpid can be either blocking or non-blocking:

- If *options* is 0, then it is blocking
- If *options* is WNOHANG, then is it non-blocking

waitpid is more flexible compared to wait:

- If *pid == -1*, it waits for any child process. In this respect, waitpid is equivalent to wait
- If *pid > 0*, it waits for the child whose process ID equals pid
- If *pid == 0*, it waits for any child whose process group ID equals that of the calling process
- If *pid < -1*, it waits for any child whose process group ID equals that absolute value of pid

wait3 **#include <sys/wait.h>**
pid_t wait3(int *stat_loc, int options, struct rusage *rusage);

wait3() and wait4():- The wait3() and wait4() system call performs the similar task to waitpid(). The core difference between them(i.e waitpid() and wait3() ,wait4()), is in terms of the resources usage information about the terminated child in the structure pointed to by the rusage argument. The information includes CPU time used by the process and memory-management statistics.

wait4 **#include <sys/wait.h>**

pid_t wait4(pid_t pid, int *stat_loc, int options, struct rusage *rusage);
The wait4() call provides a more general interface for programs that need to wait for certain child processes, that need resource utilization statistics accumulated by child processes, or that require options.

signal **#include <signal.h>**
sighandler_t signal(int *signum*, sighandler_t *handler*);
—> typedef void (*sighandler_t)(int);
A signal is a software generated interrupt that is sent to a process by the OS because of when user press ctrl-c or another process tell something to this process.
There are fix set of signals that can be sent to a process. signal are identified by integers.
Signal number have symbolic names. For example SIGCHLD is number of the signal sent to the parent process when child terminates.

Examples:
#define SIGHUP 1 /* Hangup the process */
#define SIGINT 2 /* Interrupt the process */
#define SIGQUIT 3 /* Quit the process */
#define SIGILL 4 /* Illegal instruction. */
#define SIGTRAP 5 /* Trace trap. */
#define SIGABRT 6 /* Abort. */
https://www.geeksforgeeks.org/signals-c-language/

sigaction **#include <signal.h>**
int sigaction(int sig, const struct sigaction *restrict act, struct sigaction *restrict oact);
The sigaction() system call assigns an action for a signal specified by sig. If act is non-zero, it specifies an action (SIG_DFL, SIG_IGN, or a handler routine) and mask to be used when delivering the specified signal. If oact is non-zero, the previous handling information for the signal is returned to the user.

kill **#include <signal.h>**
int kill(pid_t *pid*, int *sig*);
The kill() system call can be used to send any signal to any process group or process.

If *pid* is positive, then signal *sig* is sent to the process with the ID specified by *pid*.
If *pid* equals 0, then *sig* is sent to every process in the process group of the

calling process.
If *pid* equals -1, then *sig* is sent to every process for which the calling process has permission to send signals, except for process 1 (*init*), but see below.
If *pid* is less than -1, then *sig* is sent to every process in the process group whose ID is -*pid*.

If *sig* is 0, then no signal is sent, but existence and permission checks are still performed; this can be used to check for the existence of a process ID or process group ID that the caller is permitted to signal.

On success (at least one signal was sent), zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

exit **#include <stdlib.h>**
void exit(int status);
Terminates the calling process immediately. Any open file descriptors belonging to the process are closed and any children of the process are inherited by process 1, init, and the process parent is sent a SIGCHLD signal.

getcwd **#include <unistd.h>**
char *getcwd(char *buf, size_t size);
These functions return a null-terminated string containing an absolute pathname that is the current working directory of the calling process. The pathname is returned as the function result and via the argument *buf*, if present.

The getcwd() function copies an absolute pathname of the current working directory to the array pointed to by *buf*, which is of length *size*.

If the length of the absolute pathname of the current working directory, including the terminating null byte, exceeds *size* bytes, NULL is returned, and *errno* is set to ERANGE; an application should check for this error, and allocate a larger buffer if necessary.

On success, these functions return a pointer to a string containing the pathname of the current working directory. In the case of getcwd() and getwd() this is the same value as *buf*.

On failure, these functions return NULL, and *errno* is set to indicate the error. The contents of the array pointed to by *buf* are undefined on error.

chdir **#include <unistd.h>**
int chdir(const char *path);

chdir() changes the current working directory of the calling process to the directory specified in *path*.

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

stat **#include <sys/stat.h>**
int stat(const char *restrict path, struct stat *restrict buf);
These functions return information about a file. No permissions are required on the file itself, but-in the case of stat() and lstat() - execute (search) permission is required on all of the directories in *path* that lead to the file.

stat() stats the file pointed to by *path* and fills in *buf*.
https://linux.die.net/man/2/stat

lstat **#include <sys/stat.h>**
int lstat(const char *restrict path, struct stat *restrict buf);
lstat() is identical to stat(), except that if *path* is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

fstat **#include <sys/stat.h>**
int fstat(int fildes, struct stat *buf);
fstat() is identical to stat(), except that the file to be stat-ed is specified by the file descriptor *fd*.

unlink **#include <unistd.h>**
int unlink(const char *pathname);
unlink() deletes a name from the filesystem. If that name was the last link to a file and no processes have the file open, the file is deleted and the space it was using is made available for reuse.

If the name was the last link to a file but any processes still have the file open, the file will remain in existence until the last file descriptor referring to it is closed.
If the name referred to a symbolic link, the link is removed.
If the name referred to a socket, FIFO, or device, the name for it is removed but processes which have the object open may continue to use it.

On success, zero is returned. On error, -1 is returned, and *errno* is set to indicate the error.

execve **#include <unistd.h>**
int execve(const char *path, char *const argv[], char *const envp[]);
The exec family of functions replaces the current running process with a new process. It can be used to run a C program by using another C program.

dup **#include <unistd.h>**
int dup(int fildes);
The dup() system call creates a copy of a file descriptor.

- It uses the lowest-numbered unused descriptor for the new descriptor.
- If the copy is successfully created, then the original and copy file descriptors may be used interchangeably.
- They both refer to the same open file description and thus share file offset and file status flags.

dup2 **#include <unistd.h>**
int dup2(int fildes, int fildes2);
The dup2() system call is similar to dup() but the basic difference between them is that instead of using the lowest-numbered unused file descriptor, it uses the descriptor number specified by the user.

pipe **#include <unistd.h>**
int pipe(int fildes[2]);
Creating ``pipelines'' with the C programming language can be a bit more involved than our simple shell example. To create a simple pipe with C, we make use of the pipe() system call. It takes a single argument, which is an array of two integers, and if successful, the array will contain two new file descriptors to be used for the pipeline. After creating a pipe, the process typically spawns a new process (remember the child inherits open file descriptors).
The first integer in the array (element 0) is set up and opened for reading, while the second integer (element 1) is set up and opened for writing. Visually speaking, the output of fd1 becomes the input for fd0. Once again, all data traveling through the pipe moves through the kernel.
Remember that an array name in C *decays* into a pointer to its first member. Above, fd is equivalent to &fd[0]. Once we have established the pipeline, we then fork our new child process:
If the parent wants to receive data from the child, it should close fd1, and the child should close fd0. If the parent wants to send data to the child, it should close fd0, and the child should close fd1. Since descriptors are shared between the parent and child, we should always be sure to close the end of pipe we aren’t concerned with. On a technical note, the EOF will never be returned if the

unnecessary ends of the pipe are not explicitly closed.
https://tldp.org/LDP/lpg/node11.html

opendir **#include <dirent.h>**
DIR *opendir(const char *filename);
The **opendir**() function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream.
The stream is positioned at the first entry in the directory.

readdir **#include <dirent.h>**
struct dirent *readdir(DIR *dirp);
The **readdir**() function returns a pointer to a *dirent* structure representing the next directory entry in the directory stream pointed to by *dirp*.
It returns NULL on reaching the end of the directory stream or if an error occurred.
https://man7.org/linux/man-pages/man3/readdir.3.html

closedir **#include <dirent.h>**
int closedir(DIR *dirp);
The **closedir**() function closes the directory stream associated with *dirp*. A successful call to **closedir**() also closes the underlying file descriptor associated with *dirp*. The directory stream descriptor *dirp* is not available after this call.
The **closedir**() function returns 0 on success. On error, -1 is returned, and *errno* is set to indicate the error.

strerror **#include <stdio.h>**
char *strerror(int errnum);
The function strerror() is very similar to perror(), except it returns a pointer to the error message string for a given value (you usually pass in the variable errno.)

perror **#include <stdio.h>**
void perror(const char *s);
Print an error in a description, you can point the parameter s to it (or you can leave s as NULL and nothing additional will be printed.)

isatty **#include <unistd.h>**
int isatty(int fd);
The isatty() function determines if the file descriptor fd refers to a valid

terminal type device.
The isatty() function returns 1 if fd refers to a terminal type device; otherwise, it returns 0 and may set errno to indicate the error.

ttyname **#include <unistd.h>**
char*ttyname(int fd);
The ttyname() function gets the related device name of a file descriptor for which isatty() is true.

The ttyname() function returns the name stored in a static buffer which will be overwritten on subsequent calls.
The ttyname() function returns the null terminated name if the device is found and isatty() is true; otherwise a NULL pointer is returned.

ttyslot **#include <unistd.h>**
int ttyslot(void);
The legacy function **ttyslot**() returns the index of the current user's entry in some file.
If successful, this function returns the slot number. On error (e.g., if none of the file descriptors 0, 1, or 2 is associated
with a terminal that occurs in this data base) it returns 0 on UNIX V6 and V7 and BSD-like systems, but -1 on System V-like systems.

ioctl **#include <sys/ioctl.h>**
int ioctl(int fildes, unsigned long request, ...);
The ioctl() system call manipulates the underlying device parameters of special files. In particular, many operating
characteristics of character special files (e.g., terminals) may be controlled with ioctl() requests. The argument *fd* must be an open file descriptor.

The second argument is a device-dependent request code. The third argument is an untyped pointer to memory. It's
traditionally char **argp* (from the days before void * was valid C), and will be so named for this discussion.

An ioctl() *request* has encoded in it whether the argument is an *in* parameter or *out* parameter, and the size of the argument *argp*
in bytes. Macros and defines used in specifying an ioctl() *request* are located in the file *<sys/ioctl.h>*. See NOTES.

Usually, on success zero is returned. A few ioctl() requests use the return value as an output parameter and return a nonnegative
value on success. On error, -1 is returned, and errno is set to indicate the error.

getenv **#include <stdlib.h>**

char *getenv(const char *name);

The getenv() function obtains the current value of the environment variable, name. The application should not modify the string pointed to by the getenv() function.

The getenv() function returns the value of the environment variable as a NULL-terminated string. If the variable name is not in the current environment, NULL is returned.

tcsetattr **#include <termios.h>** **#include <unistd.h>**

int tcsetattr(int *fd*, int *optional_actions*, const struct termios **termios_p*);

The tcgetattr() function copies the parameters associated with the terminal referenced by fildes in the termios structure referenced by termios_p.

This function is allowed from a background process; however, the terminal attributes may be subsequently changed by a foreground process.

tcgetattr **#include <termios.h>** **#include <unistd.h>**

int tcgetattr(int *fd*, struct termios **termios_p*);

tcgetattr() gets the parameters associated with the object referred by *fd* and stores them in the *termios* structure referenced by *termios_p*. This function may be invoked from a background process; however, the terminal attributes may be subsequently changed by a foreground process.

tgetent **#include <curses.h>** **#include <term.h>**

int tgetent(char *bp, const char *name);

These routines are included as a conversion aid for programs that use the *termcap* library. Their parameters are the same and the routines are emulated using the *terminfo* database. Thus, they can only be used to query the capabilities of entries for which a terminfo entry has been compiled.

The tgetent routine loads the entry for *name*. It returns 1 on success, 0 if there is no such entry, and -1 if the terminfo database could not be found. The emulation ignores the buffer pointer *bp*.

https://linux.die.net/man/3/tgetent

tgetflag **#include <curses.h>** **#include <term.h>**

int tgetflag(char *id);

The tgetflag routine gets the boolean entry for *id*, or zero if it is not available.

tgetnum **#include <curses.h>** **#include <term.h>**
int tgetnum(char *id);
The tgetnum routine gets the numeric entry for *id*, or -1 if it is not available.

tgetstr **#include <curses.h>** **#include <term.h>**
char *tgetstr(char *id, char **area);
The tgetstr routine returns the string entry for *id*, or zero if it is not available. Use tputs to output the returned string. The return value will also be copied to the buffer pointed to by *area*, and the *area* value will be updated to point past the null ending this value.

tgoto **#include <curses.h>** **#include <term.h>**
char *tgoto(const char *cap, int col, int row);
The tgoto routine instantiates the parameters into the given capability. The output from this routine is to be passed to tputs.

tputs **#include <curses.h>** **#include <term.h>**
int tputs(const char *str, int affcnt, int (*putc)(int))
The tputs routine applies padding information to the string *str* and outputs it. The *str* must be a terminfo string variable or the return value from tparm, tgetstr, or tgoto. *affcnt* is the number of lines affected, or 1 if not applicable. *putc* is a putchar-like routine to which the characters are passed, one at a time.